VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SLEDOVÁNÍ POHYBU V MHD

MOVEMENT TRACKING IN PUBLIC TRANSPORT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JÁN PAULOVČÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20065/2016/xpaulo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Paulovčák Ján, Bc.**
Obor: Informační systémy
Téma: **Sledování pohybu v MHD**
Movement Tracking in Public Transport
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s dostupnými službami IDS JMK pro zjišťování polohy zastávek a dopravních prostředků.
2. Prostudujte existující technologie pro tvorbu aplikací klient-server s mobilním klientem na platformě Android.
3. Navrhněte prototyp aplikace pro sledování pohybu uživatele v síti MHD v Brně a optimalizaci trasy a přestupů při cestování.
4. Po dohodě s vedoucím implementujte serverovou část pomocí vhodných technologií a klientskou část na platformě Android.
5. Proveďte testování vytvořeného řešení v reálném prostředí.
6. Zhodnoťte dosažené výsledky a navrhněte možná další rozšíření.

Literatura:

- Lacko, L.: Vývoj aplikací pro Android, Computer Press, 2015
- Juneau, J.: Java EE 7 Recipes, Apress, 2013
- Žára, O.: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Burget Radek, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto diplomová práca sa zaoberá problematikou sledovania pohybu užívateľa v mestskej hromadnej doprave a taktiež optimalizáciou trasy a prestupov pri cestovaní. V jej prvej časti autor prezentuje informácie o teórii grafov a príslušných algoritmoch, konkrétne Slepé prehľadávanie do šírky, Djikstrov algoritmus a algoritmus A*. V druhej časti autor popisuje návrhv serverovej a mobilnej aplikácie s popisom procesu plánovania a ovládania jednotlivých častí.

Abstract

This master's thesis is dedicated to user's movement tracking in public transport, as well as to optimize route planning. In the first part, author presents information related to graph theory and graph theory algorithms, including Breadth-First Search, Djikstra's algorithm and A*. In the second part of this thesis, author describes the design of server and mobile application including description of routing process and how individual parts works.

Kľúčové slová

grafy, plánovanie trasy, algoritmus, Android, BFS, Djikstra, A*, Java, MongoDB, mobilná aplikácia, mestská hromadná doprava

Keywords

graphs, route planning, algorithm, Android, BFS, Djikstra, A*, Java, MongoDB, mobile application, public transport

Citácia

PAULOVČÁK, Ján. *Sledování pohybu v MHD*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Burget Radek.

Sledování pohybu v MHD

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Radeka Burgeta, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Ján Paulovčák
22. mája 2017

Poďakovanie

Ďakujem pánovi Ing. Radekovi Burgetovi Ph.D. za ochotu pri výbere témy a vedení tejto diplomovej práce, za ľudský prístup, cenné pripomienky a pomoc so sprístupnením externých dát. Vďaka taktiež patrí aj firme Kordis JMK a.s., ktorá poskytla dáta potrebné k realizácii tejto práce.

Obsah

Kapitola 1

Úvod

V dnešnej dobe sa infraštruktúra ciest a spôsob cestovania stáva čím ďalej, tým viac komplexnejšou témou. Spolu s rozrástajúcimi sa dopravnými sieťami sa zvyšuje aj doležitosť a schopnosť mobility ľudí v súčasnej spoločnosti. Avšak, zložitejšie siete, veľky počet dopravných prostriedkov, množstvo možností ako sa dopraviť z bodu A do bodu B, automaticky neznamená zjednodušenie cestovania pre cestujúcich. Práve naopak, plánovanie takejto trasy sa pre nich stáva komplexnejšou a náročnejšou problematikou. V dôsledku toho vzniká dopyt po lepších a efektívnejších metódach plánovania trasy.

Jeden z prvých automatizovaných spôsobov ako uľahčiť ľudom cestovanie boli navigačné systémy, ktoré využívali svoje vlastné metódy plánovania trasy a boli určené najmä pre cestnú dopravu s využitím áut. Spoločným problémom, resp. obmedzením týchto zariadení bolo, že umožňovali plánovanie trasy len v rámci ich vlastnej domény, a teda nedokázali kombinovať rôzne spôsoby dopravy.

Proces plánovania trasy pre autá a mestskú hromadnú dopravu je značne rozdielny. Kým sieť ciest pri plánovaní trasy určenej pre autá môže byť využívaná v ľubovoľnom čase, v prípade mestskej hromadnej dopravy je potrebné počítať s časmi odchodov a príchodov jednotlivých vozidiel. Spoločným konceptom však zostáva namodelovať obe siete ako orientovaný graf. Pre sieť ciest využívanej k plánovaniu trasy pre autá budú križovatky tvoriť uzly grafu a cesty budú zobrazovať hrany grafu. Cenou hrany potom bude čas potrebný k prejdeniu daného úseku cesty. Najrýchlejšia cesta z bodu `A` do bodu `B` je potom vypočítaná napríklad pomocou `Djiktstrovho algoritmu`. V prípade mestskej hromadnej dopravy môžeme základný problém plánovania trasy formulovať ako: Pre danú cieľovú a počiatočnú stanicu, s daným časom odchodu z počiatočnej stanice, aká je najlepšia kombinácia jednotlivých spojov tak, aby sme sa z počiatočnej stanice dostali do cieľovej stanice v čo najkratšom čase?

1.1 Cieľ diplomovej práce

Cieľom tejto práce je vytvoriť systém, ktorý dokáže sledovať pohyb užívateľa v MHD a optimalizuje plánovanie trasy a prestupov v rámci mestskej hromadnej dopravy v Brne. Súčasťou tohto systému je aj mobilná aplikácia pre platformu Android, ktorá dokáže užívateľovi naplánovať čo najlepšiu trasu a v prípade potreby túto trasu zmeniť, resp. preplánovať. Prínos tejto práce bude hlavne pre koncových užívateľov mobilnej aplikácie, teda najmä ľudí, ktorí využívajú mestskú hromadnú dopravu v Brne. Nakoľko je aplikácia určená širokému spektru užívateľov, jej ovládanie by malo byť jednoduché a intuitívne.

1.2 Členenie práce

Práca je rozčlenená do desiatich rôznych kapitol popísujúcich problémy a návrhy riešení pre daný problém. Kapitola 2 je venovaná oboznámeniu s poskytovateľom externých dát, ktoré sú potrebné k technickej realizácii tejto práce. V kapitole 3 sa zaoberám teoretickými poznatkami z oblasti grafov a riešenia problémov, ktoré sú s touto témou spojené. V kapitole 4 venujem pozornosť popisu jednotlivým grafovým algoritmom spolu s vysvetlením princípu ich fungovania a zhodnotením výhod a nevýhod, ktorými disponujú. Architektúru aplikácie popisujem v kapitole 5 a zvolené technológie spolu s celkovým návrhom serverovej časti v kapitole 6. Kapitola 7 je venovaná návrhu klientskej časti - mobilnej aplikácie spolu s detailným popisom jednotlivých prvkov, ako aj vysvetlením ako tieto prvky medzi sebou interagujú. V kapitole 8 sa zaoberám implementáciou serverovej a klientskej časti. V predposlednej kapitole sú opísané spôsoby a výsledky testovania. Posledná desiata kapitola je venovaná možným rozšíreniam.

Kapitola 2

Kordis JMK

Ako už bolo spomenuté v úvodnej kapitole, cieľom tejto diplomovej práce je vytvorenie systému, ktorý optimalizuje a zvýši efektivitu cestovania pomocou mestskej hromadnej dopravy mesta Brna. K dosiahnutiu tohto cieľa je však potreba práce s dátami, ktorých získanie značne presahuje možnosti študenta, a to nielen po finančnej, ale aj časovej stránke. Ide najmä o dáta, ktoré obsahujú polohy jednotlivých zastávok v meste, informácie o linkách, ale aj časové záznamy a to v podobe príjazdov a odjazdov spojov z jednotlivých zastávok. K získaniu takýchto dát bola oslovená firma `Kordis JMK, a.s`, ktorá pôsobí v Brne už od roku 2002, a ktorej jednou z činností je aj prevádzkovanie integrovaného dopravného systému na území Jihomoravského kraja. Tá okrem iného trvalo sleduje a vyhodnocuje vývoj prepravných potrieb, optimalizuje vedenie liniek a podieľa sa na vytváraní cestovných poriadkov[1]. Oslovená firma Kordis JMK súhlasila s poskytnutím takýchto dát, a to nielen statických; teda polohy, názvy a popisy jednotlivých zastávok, liniek či časových záznamov potrebných k plánovaniu trasy, ale aj dynamických akými sú aktuálne polohy vozidiel či meškania vozidiel, prípadne iné typy informácií, ktoré sú špecifické na základe rôznych faktrov. Vďaka takýmto informáciám je možné realizáciu tejto diplomovej práce posunúť o krok ďalej. Tie sú dostupné z dvoch zdrojov.

Prvým zdrojom je `FTP` server, ktorý obsahuje najmä informácie o jednotlivých zastávkach (identifikátor, zóna, názov, GPS súradnice jednotlivých stĺpikov, popis stĺpikov a pod.), prípojoch a cestovných poriadkov. Aktualizovaný je približne každých 5-7 dní. Druhým zdrojom je webová služba, ktorá sprostredkováva aktuálne informácie okrem iného aj o:

- **vozidlách** - kód linky a vozidla, GPS súradnice, identifikátor predchádzajúcej zastávky, typ vozidla (nízkopodlažné), aktuálne meškanie a pod.
- **odjazdoch zo zastávok** - najbližšie možné odjazdy jednotlivých vozidiel z konkrétnej zastávky, typ zastávky, koncová zastávka pre daný odjazd a pod.
- **štatistikách** - aktuálne štatistické údaje - percentuálny podiel meškajúcich vozidiel MHD, vlakov, a pod.
- **elektrických paneloch, zastávkach, ...**

[1] http://idsjmk.cz/onas.aspx

Kapitola 3

Grafy

Táto kapitola je venovaná popisu teoretickej časti grafov, vysvetleniu základných pojmov a rozdelení, ktoré sú potrebné k pochopeniu nasledujúcich kapitol. Predstavený je taktiež jeden zo základných problémov v oblasti grafov.

3.1 Teória grafov

Množstvo problémov v skutočnom živote by mohlo byť s jednoduchosťou vyobrazených, respektíve popísaných nejakým diagramom, ktorý pozostáva z niekoľkých bodov, a tieto body sú istým spôsobom navzájom pospájané čiarami. Príkladom jednotlivých bodov by mohli byť ľudia pričom čiary by reprezentovali určitú formu vzťahu medzi nimi. Alebo elektrická sieť, kde body tvoria stĺpy a čiary spajajúce tieto body sú elektrické káble. V kontexte takýchto diagramom nás v konečnom dôsledku nezaujíma dôvod alebo význam čo jednotlivé spojenia reprezentujú, ale či sú dané dva body navzájom spojené alebo nie. V matematickom ponímaní sa takýto diagram nazýva grafom.

Grafy sú časťou diskrétnej matematiky a sú silným nástrojom, ktorý umožňuje modelovať vzťahy medzi objektami. Prvé štúdie o grafoch siahajú až do 18. storočia, kedy bol definovaný matematický problém siedmych mostov v meste Königsberg v Prusku (dnešný Kaliningrad, Rusko). Otázka dnes už vyriešeného problému spočívala v tom, či je možné prejsť všetkých sedem mostov, ktoré spájali dva ostrovy s mestom tak, aby ten, kto sa o to pokúša, vstúpil na každý jeden most práve raz. Grafy môžu byť taktiež využité pri modelovaní problémov v mnohých oblastiach, ako napríklad transport, plánovanie, robotika, siete a pod. Množstvo optimalizačných problémov z takýchto oblastí možu byť preformulované práve do oblasti teórie grafov.

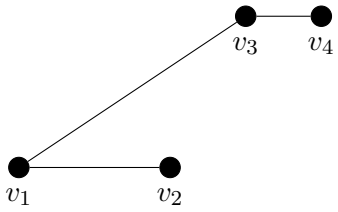
Za problémami spojenými so smerovaním alebo plánovaním trasy sa nachádza koncept `minimálnej cesty` - teda vytvorenie cesty v rámci danej siete (grafu) z nejakého vopred známeho počiatočného bodu do ľubovoľného koncového bodu nachádzajúceho sa v rovnakej sieti so zreteľom na čo najmenšiu vzdialenosť, respektíve nejakú inú veličinu spojenú so vzidalenosťou, ako napríklad čas potrebný k prejdeniu danej cesty. Toto je jedným zo základných problémov v teórii grafov.

V koncepčnom vnímaní je graf tvorený uzlami a hranami spájajucími dané uzly. Formálne je však graf reprezentovaný ako množina uzlov a hrán $G = (V, E)$, kde V je množina uzlov (angl. `vertices`) a E množina hrán (angl. `edges`). Každá hrana je tvorená dvojicami uzlov $E \subseteq V \times V$. Hovoríme, že hrana z $u \in V$ do $v \in V$ existuje vtedy, a len vtedy

ak $(u, v) \in E$. Ak $(u, v) \in E$ je hranou grafu $G = (V, E)$ hovoríme, že tieto vrcholy sú susedné.[26]

Existuje niekoľko typov grafov podľa hrán, ktoré obsahujú:

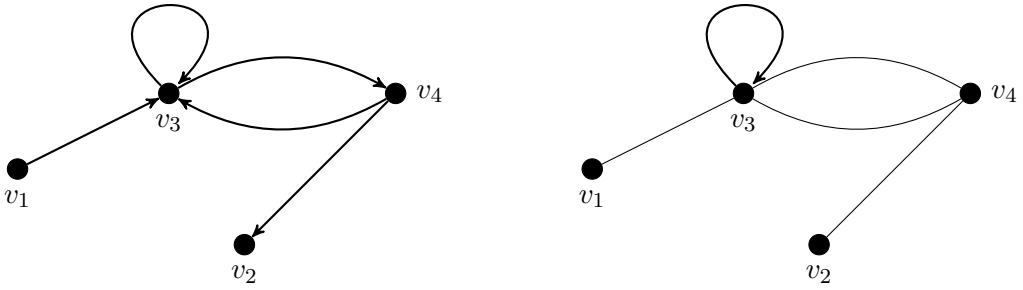
- **Neorientované** - neorientovaný graf $G = (V, E)$ tvorí množina vrcholov V a množina hrán E tak, že každá hrana $e \in E$ je priradená **neusporiadanej** dvojici vrcholov $u, v \in V$.[17] Takéto hrany niesú orientované, respektíve, každá takáto hrana je orientovaná oboma smermi. V prípade, že je jednej dvojici vrcholov priradených viacero hrán sa tieto hrany nazývajú násobné. Príkladom štruktúr využívajúcich neorientované grafy môžu byť napríklad stromy a bipartitné grafy.



Obr. 3.1: Neorientovaný graf

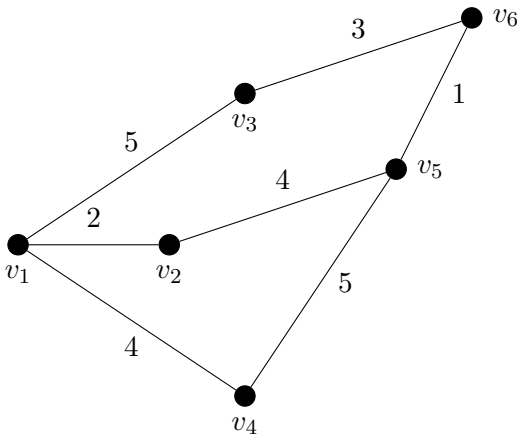
- **Orientované** - v niektorých prípadoch na vyjadrenie určitého problému jednoduché neorientované grafy nestačia. Napríklad v prípade modelovania siete ciest je potrebné vedieť nielen to, medzi ktorými dvoma miestami cesta vedie, ale aj to ako je tá cesta orientovaná. Príkladom môžu byť jednosmerné cesty, ktoré umožňujú prejsť autám len v určitom smere. Z tohto príkladu je jasné, že jednoduché grafy nám postačovať nebudú, a potrebujeme graf, v ktorom každá hrana má určený smer. Podobne ako v prípade neorientovaných grafov sú tieto grafy tvorené z množiny vrcholov V a množiny hrán E tak, že každá hrana $e \in E$ je priradená **usporiadanej** dvojici vrcholov $u, v \in V$.[17] Ku každému orientovanému grafu D môžeme vytvoriť graf G s rovnakou množinou vrcholov a množinou hrán, pre ktorú platí, že ku každej hrane grafu D existuje hrana grafu G, ktorá končí v rovnakých vrcholoch ako hrana grafu D. Takýto graf G sa potom nazýva základný graf. A naopak, ku každému neorientovanému grafu môžeme vytvoriť orientovaný graf tak, že každej z hrán priradíme orientáciu na jej konci.[6] Orientácia hrany je väčšinou zobrazovaná pomocou šípky. Dôležitou podtriedou takýchto orientovaných grafov, ktoré sa využívajú v mnohých oblastiach sú orientované acyklické grafy.

- **Ohodnotený graf** - takéto grafy sú tvorené z množiny vrcholov V a množiny hrán E, pričom majú každú zo svojich hrán ohodnotenú príslušnou cenou. To znamená, že s presunom z jedného vrcholu do niektorého z jeho susedných vrchlovov, je spojená istá cena. Táto cena potom reprezentuje ohodnotenie hrany medzi takýmito dvoma vrcholmi. Cenu može reprezentovať napríklad čas potrebný k dosiahnutiu susedného vrcholu, priepustnosť medzi dvoma susednými vrcholmi, rýchlosť a pod.
  Nech $c : E \to R$ je zobrazením, potom (G, c) je ohodnotený graf.



(a) orientovaný graf (b) základný graf

Obr. 3.2: Orientovaný graf spolu so základným grafom



Obr. 3.3: Ohodnotený graf

3.2 Problém minimálnej cesty

„Cesta dlhá tisíc míľ začína prvým krokom“ (Confucius)

Jedným z dôležitých problémov spojených s grafmi je problém minimálnej cesty. Tento problém spočíva v nájdení minimálnej cesty medzi dvoma ľubovoľnými bodmi vrámci daného grafu. Príkladom takéhoto problému môže byť nájdenie minimálnej cesty medzi dvoma bodmi na mape ciest. Nech graf $G = (V, E)$ je ohodnotený graf a $c : E \to R$ je funkciou ohodnotenia hrany. Potom cesta z vrcholu v do vrcholu w je cesta $P = (v_1, \ldots, v_n)$, $v_1 = v$, $v_n = w$. Výslednou cenou cesty P je suma jednotlivých cien hrán v ceste P, $c(P)$

$$c(P) = \sum_{i=1}^{n-1} c(e_{i,i+1});\ e_{i,i+1} = (v_i, v_{i+1}) \tag{3.1}$$

Najmenšia hodnota $c(P)$ pre všetky cesty z vrcholu v do vrcholu w je potom označovaná ako $\delta(u, v)$. Minimálna cesta z vrcholu v do vrcholu w je cesta P, pričom platí, že $c(P) = \delta(u, v)$. V prípade, že každá z hrán grafu je ohodnotená rovnakou cenou $c : E \to 1$, je nájdená minimálna cesta zároveň cestou s najmenším počtom hrán. Implementácie algoritmov, ktoré riešia tento problém viedli k vytvoreniu nových dátových štruktúr ako `Disjoint-Set` alebo `Fibonacci Heap`.[16]

Existuje niekoľko problémov, ktoré sú variáciami problému minimálnej cesty. Ich názvy však zostávajú v pôvodnom anglickom znení kvôli možným nejastnostiam v preklade:

- **Single-source shortest-paths problem** - tento problém spočíva v nájdení minimálnej cesty v grafe $G = (V, E)$ medzi počiatočným vrcholom $v \in V$ a každým ďalším vrcholom z množiny V
- **Point-to-Point shortest-path problem** - ide o problém nájdenia minimálnej cesty v grafe $G = (V, E)$ medzi dvoma konkrétnymi vrcholmi $v \in V$ a $w \in V$. V prípade, že `Single-source shortest-paths problem` s počiatočným vrcholom v je vyriešený, potom aj tento problém je vyriešený. Ide o špecifický prípad problému `SSSP`.
- **All-pairs shortest-paths problem** - v tomto prípade pre graf $G = (V, E)$ sú namiesto konkrétných vrcholov $v \in V$ a $w \in V$ dané množiny počiatočných vrcholov S a koncových vrcholov T. Tieto množiny sú totožné s množinou vrcholov V v grafe G. Vyriešenie tohto problému automaticky zahŕňuje vyriešené problémy pre všetky variácie `problému minimálnej cesty`. Avšak, vyriešenie tohto problému je značne časovo a pamäťovo náročné.

Kapitola 4

Grafové algoritmy

Grafové algoritmy majú veľké množstvo využití. Predstavte si situáciu, že ste predavač, ktorý distribuje svoj tovar do okolitých miest. K nájdeniu čo najoptimálnejšej (v tomto prípade najlacnejšej z pohľadu dopravy) trasy je potrebné efektívne vyhľadať cestu v ohodnotenom grafe, kde jednotlivé vrcholi korešpondujú s mestami a cena hrany medzi dvoma vrcholmi (mestami) značí cenu za dopravu, ktorú potrebujete zaplatiť.

Algoritmy, ktoré riešia problém minimálnej cesty patria medzi jedny z najdôležitejších algoritmov v teórii grafov vôbec. V tejto kapitole si predstavíme niekoľko základných algoritmov, ktoré dokážu prechádzať grafom a taktiež uvedieme základné dátové štruktúry, ktoré tieto algoritmy používajú.

4.1 Algoritmus slepého prehľadávania do šírky

Prvým z popisovaných algoritmov, ktorý umožňuje prechádzať grafom je Breadth First Search.[20] Ide o algoritmus, ktorý bol predstavený profesorom Edwardom F. Moorom, v roku 1959 v kontexte hľadania najkratšej cesty vonku z bludiska. Neskôr bol nezávisle objaveným v roku 1961 C.Y.Leeom.[18] Princíp algoritmu je možné popísať nasledujúcim spôsobom. Počnúc počiatočným vrcholom $s \in V$ z grafu $G = (V, E)$ preskúmame jeho okolie, a to tak, že nájdeme všetky vrcholy, ktoré sú bezprostrednými následovníkmi vrcholu v. Rovnaký postup je potom aplikovaný na všetky vrcholy grafu G. Výsledkom je stromová štruktúra`(breadth-first tree)`, ktorej koreňom je vrchol s a tento strom je podgrafom grafu G. V prvom kroku strom obsahuje len koreň, ktorým je počiatočný vrchol. V prípade, že algoritmus pri prehľadávaní bezprostredných následnovníkov vrcholu u nájde nejaký vrchol v, ktorý doposiaľ nebol objavený, vrchol v a hrana (u, v) budú pridané do stromu. Nakoľko vrchol môže byť objavený len raz, každý z nich má najviac jedného rodiča.[7]

Algoritmus preskúmava okolie bodu rovnomerne, pričom môže byť použitý k riešeniu rôznych problémov ako napríklad `Garbage collection` alebo zisťovaniu bipartitnosti grafu. `Breadth First search` môže byť použitý ako na orientované, tak aj na neorientované grafy. Pseudokód tohto algoritmu je zobrazený v algoritme 1.

K svojmu výpočtu využíva špeciálnu dátovú štruktúru podobnú listu, nazývanú `fronta`. Táto dátova štruktúra je podobná fronte v skutočnom svete, kde objekt, ktorý do fronty vstúpil ako prvý, bude obslúžený ako prvý. Objekt, ktorý bol do fronty zaradený po ňom, bude obslúžený ako druhý a podobne. Formálne je fronta podobná listu, a teda dĺžka resp. veľkosť fronty je rovná počtu prvkov, ktoré obsahuje. Operácia pri ktorej zaraďujeme nový prvok do fronty sa nazýva `enqueue`. Takto pridaný prvok je zaradený na koniec fronty.

Algoritmus 1: Algoritmus slepého prehľadávania do šírky

Input: A graph $G = (V, E)$ and starting vertex $s \in V$.

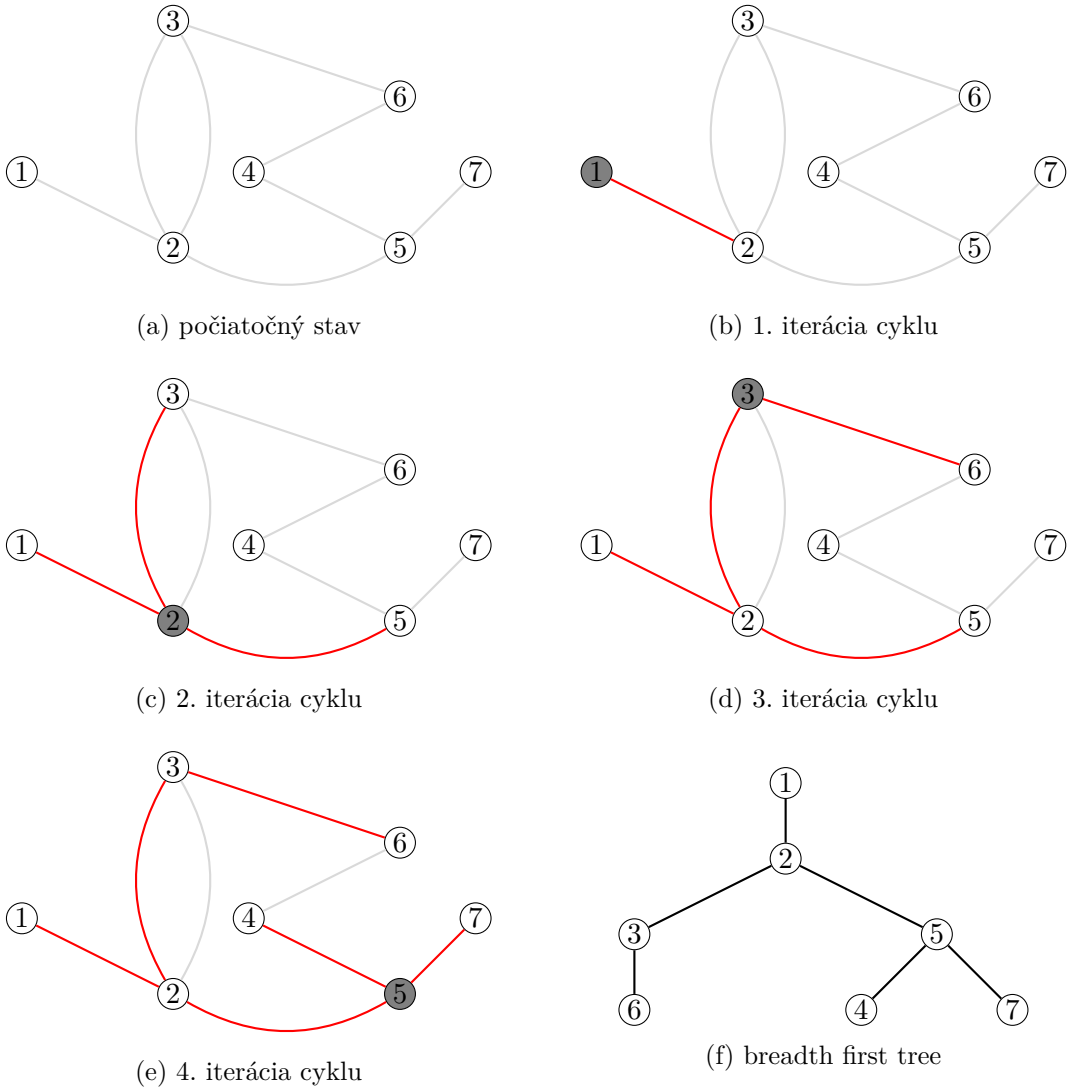
```
1   Q = Queue()
2   forall v in graph.vertexes do
3       v.distance = ∞
4       v.parent = nil
5   end
6
7   s.distance = 0
8   Q.enqueue(s)
9   while not Q.isEmpty() do
10      current = Q.dequeue()
11      forall adj in graph.adjacent(current) do
12          if adj.distance == ∞ then          /* vertex not visited yet */
13              adj.distance = current.distance + 1
14              adj.parent = current
15              Q.enqueue(adj)
16      end
17  end
```

Naopak, operácia, pomocou ktorej prvok z fronty vyberieme sa nazýva `dequeue`. Vo fronte je stále prístupný len prvý prvok, a teda fronta používa resp. implementuje politiku FIFO (`First-In, First-Out`).

V prvom kroku algoritmus inicializuje všetky vrcholy grafu, a to tak, že vzdialenosť k danému vrcholu nastaví na nekonečno, čo indikuje, že daný vrchol nebol objavený a odkaz na rodičovský vrchol nastaví na `nil`. Následne nastaví počiatočnému vrcholu vzdialenosť na hodnotu 0 a zaradí počiatočný vrchol do fronty. Potom vykonáva telo cyklu, až kým fronta vrcholov na spracovanie nebude prázdna, pričom v každej z iterácii nájde bezprostredných následovníkov práve spracovaného vrcholu a v prípade, že daný následovník ešte nebol spracovaný (hodnota `distance` je rovná ∞), nastaví tomuto vrcholu vzdialenosť o 1 väčšiu než má práve spracovávaný vrchol. Ďalej nastaví rodiča na práve spracovávaný vrchol a zaradí tohto následovníka na koniec fronty.

Aj napriek svojmu prínosu pre isté formy problémov a možnosti prechádzať grafom napríklad s cieľom nájdenia cesty k istému vrcholu vrámci grafu spolu s počtom hrán, ktoré k danému vrcholu vedú, nieje tento algortimus vhodný pre hľadanie minimálnej cesty nakoľko nezahŕňa do svojich výpočtov ohodnotenia hrán.

Nasledujúci príklad na obrázku 4.1 demonštruje postup algoritmu pri neorientovanom grafe.



(a) počiatočný stav

(b) 1. iterácia cyklu

(c) 2. iterácia cyklu

(d) 3. iterácia cyklu

(e) 4. iterácia cyklu

(f) breadth first tree

Obr. 4.1: Priebeh metódy Breadth First Search

4.2 Dijkstrov algoritmus

Dijkstrov algoritmus[8] je ďalším z algoritmov, ktorý umožňujú prechádzať grafom. Objavený bol holandským informatikom Edsgerom Wybe Dijkstrom v roku 1959. Algoritmus rieši problém minimálnej cesty(`Single-source shortest-paths problem`) v ohodnotenom grafe. Príkladom môže byť problém nájdenia najkratšej trasy na mape ciest, kde vrcholy reprezentujú mestá a cesty sú vyobrazené hranami. Použitím tohto algoritmu dokážeme nájsť najkratšiu cestu z nejakého určeného mesta do ľubovoľného iného mesta na mape. Nevýhodou, respektíve obmedzením tohto algoritmu je, že narozdiel od `Bellman-Ford` algoritmu[3][15], nedokáže pracovať s hranami ohodnotenými zápornou hodnotou. Na druhej strane, so správnou implementáciou je čas potrebný v výpočtu problému pomocou Dijkstrovho algoritmu lepší[8]. K svojmu fungovaniu využíva dátovú štruktúru s názvom `prioritná fronta`, ktorá je modifikovanou obdobou základnej verzie `fronty`. Princíp takejto fronty spočíva v zaraďovaní nových prvkov do fronty na základe priority aká je s daným prvokm spojená. V tomto prípade využívame frontu s minimálnou prioritou, pri ktorej je

nový prvok vložený do fronty tak, aby bol čím bližšie k začiatku fronty, a súčasne „nepredbehol“ žiaden prvok s nižšou prioritou. Takto uložené prvky vo fronte zostávajú zoradené podľa priority, a teda asymptotická zložitosť jednotlivých operácií nad touto frontou sa pohybuje medzi $O(1)$ a $O(\log n)$ na základe dátovej štruktúry, ktorá bola k implementácií použitá. Priorita jednotlivých prvkov je v tomto prípade ekvivalentná s cenou, resp. vzdialenosťou, ktorá je potrebná k dosiahnutiu daného vrcholu. Algoritmus je zovšeobecnením algoritmu `Breadth First search` a za podmienky, že by každá z hrán daného grafu bola ohodnotená rovnakou cenou, obe algoritmy by principiálne pracovali rovnako. V takomto prípade by bolo použitie `BFS` lepšie nakoľko Dijkstrov algoritmus používa k uchovávaniu vrcholov prioritnú frontu, ktorej asymptotická zložitosť niektorých operácií je logaritmická $O(\log n)$, kdežto pri `BFS` je využívaná jednoduchá fronta, ktorej operácie majú konštantnú asymptotickú zložitosť $O(1)$.

Pseudoalgoritmus popisujúci princíp Dijkstrovej metódy je zobrazený v algoritme 2.

Algoritmus 2: Dijkstrov algoritmus

Input: A weighted graph $G = (V, E)$ and starting vertex $s \in V$.

```
1   Q = MinPriorityQueue()
2   forall v in graph.vertexes − {s} do
3       v.distance = ∞
4       v.parent = nil
5       Q.enqueue(v.distance, v)
6   end
7
8   s.distance = 0
9   s.parent = nil
10  Q.enqueue(0, s)
11  while not Q.isEmpty() do
12      current = Q.dequeue()          /* dequeue item with lowest priority */
13      forall adj in graph.adjacent(current) do
14          if Q.contains(adj) then
15              cost = current.distance + weight(current, adj)
16              if cost < adj.distance then          /* path has better cost */
17                  adj.distance = cost
18                  adj.parent = current
19                  Q.decreasePriority(cost, adj)
20      end
21  end
```

V inicializačnej časti algoritmus nastaví vzdialenosť všetkým vrcholom okrem počiatočného na kladné nekonečno, odkaz k rodičovi na `nil` a zaradí ich do fronty. Následne inicializuje a zaradí do fronty aj počiatočný vrchol a to nastavením jeho vzdialenosti, a teda aj priority na 0, čím vynúti to, aby bol tento vrchol spracovaný v cykle ako prvý. Cyklus na riadku `11` následne spracúvava jednotlivé položky vo fronte až kým fronta nie je prázdna, a to tým spôsobom, že vyberie a odstráni z fronty vrchol s najmenším ohodnotením (vzdialenosťou), nájde všetkých jeho bezprostredných následovníkov, ktorí sú zároveň ešte vo fronte. Pre týchto svojích následovníkov vypočíta vzdialenosť, ktorá je potrebná k ich dosiahnutiu a v prípade, že je táto vzdialenosť menšia než tá, ktorú majú v danej

chvíli uloženú, aktualizuje túto hodnotu a zmení odkaz na rodiča na súčasný vrchol. Taktiež danému následovníkovi aktualizuje prioritu vo fronte na novú hodnotu. Až cyklus skončí, celý graf bol spracovaný a dokážeme určiť vzdialenosť z počiatočného vrcholu k ľubovoľnému inému vrcholu vrámci daného grafu.

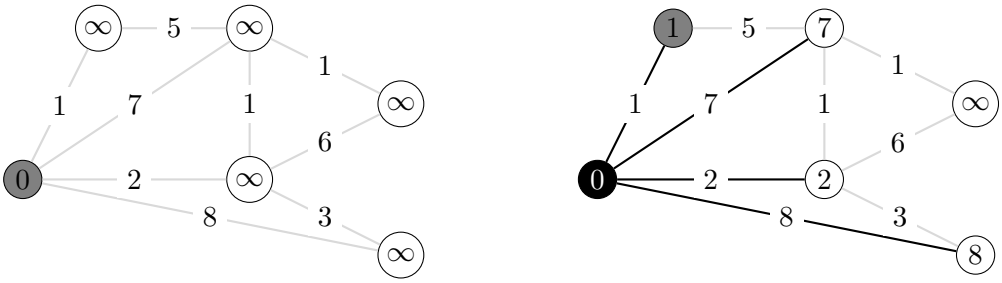
Optimalizáciou tohto algoritmu v prípade, že hľadáme vzdialenosť medzi dvoma presnými vrcholmi môže byť zaradenie ukončujúcej podmienky za riadok `12`,
ktorá v prípade, že sa `current` vrchol rovná hľadanému vrcholu cyklus predčasne ukončí.

Algoritmus 3: Optimalizácia Dijkstrovho algoritmu ukončujúcou podmienkou

```
1    if current == target then
2        break
```

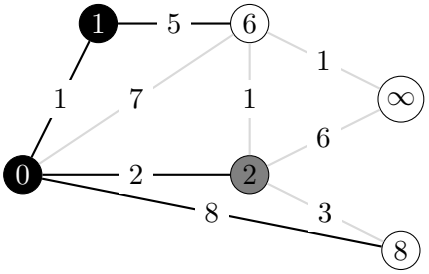
Dôležité však je umiestniť túto ukončujúcu podmienku presne do tohto miesta, a nie do časti, kde na daný vrchol narazíme vrámci prehľadávania bezprostredných následovníkov práve spracovávaného vrcholu (za riadok `13`). V taktomto prípade by algoritmus mohol skončiť predčasne a výsledná cesta by nemusela byť minimálnou.

Séria grafov na obrázku 4.2 zobrazuje princíp fungovania Dijkstrovho algoritmu.

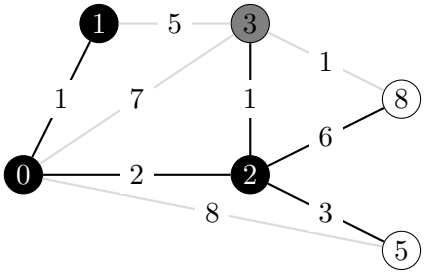


(a) počiatočný stav

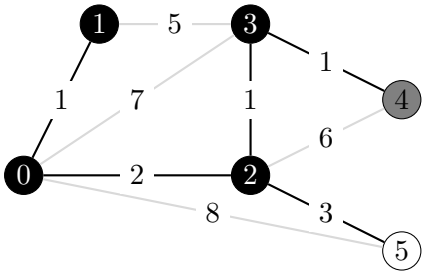
(b) 1. iterácia cyklu



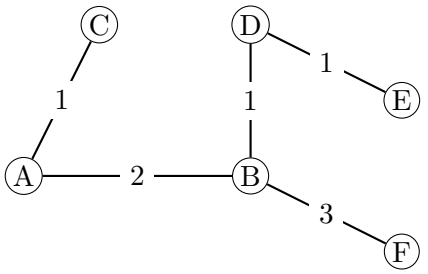
(c) 2. iterácia cyklu



(d) 3. iterácia cyklu



(e) 4. iterácia cyklu



(f) výsledný podgraf

Obr. 4.2: Priebeh Dijkstrovho algoritmu na ohodnotenom grafe

Posledné dve iterácie pre uzly s ohodnotením 4 a 5 nie sú v obrázku 4.2 zobrazené, nakoľko žiaden z ich už nemá nejakého bezprostredného následovníka, ktorý by súčasne bol vo fronte, a teda nič iné sa už pri prechode grafom nemôže zmeniť. Pre priehľadnosť a lepšie pochopenie princípu algoritmu sú v obrázku jednotlivé vrcholy farebne oddelené a každý z vrcholov má zobrazenú aktuálnu vzdialenosť od počiatočného vrcholu. Šedou farbou je označovaný vrchol, ktorý má najnižšiu vzdialenosť od počiatočného vrcholu (prioritu v rámci fronty), a teda sa vo fronte nachádza na začiatku. Čiernou farbou sú zasa označené vrcholy, ktoré už boli spracované a vyhodené z fronty. V prípade, že je medzi dvoma vrcholmi hrana aktívna, je zvýraznená tmavou farbou, naopak neaktívne hrany sú svetlo šedé. Na poslednom grafe je možné vidieť výsledný podgraf, ktorý zobrazuje najkratšiu cestu k jednotlivým vrcholom od počiatočného vrcholu.

4.3 Algoritmus A*

Posledným z popisovaných algoritmov, ktoré sú schopné prechádzať grafom je skupina, respektíve rodina algoritmov A*[14]. Ide o najznámešiu a najpoužívanejšiu metódu v oblasti prehľadávania stavového priestoru[11]. Objavený bol troma pánmi zo skupiny `Stanford Research Institute` v roku 1968. Ide o modifikovanú verziu Dijkstrovho algoritmu, popisovaného v kapitole 4.2, ktorá využíva k svojím výpočtom heuristiku pre zvýšenie efektivity prehľadávania. Typickým príkladom môže byť vyhľadávanie najkratšej trasy medzi dvoma mestami, pričom v prospech vyhľadávania môžeme využiť znalosť domény. Napr. znalosť, že žiadna vzdialenosť s použitím ciest medzi dvoma mestami nemôže byt kratšia ako ich vzdušná vzdialenosť. V priebehu expandovania jednotlivých vrcholov je možné uchovávať si najmenšiu cenu akou je možné sa k danému vrcholu dostať, ale aj odkaz k rodičovskému vrcholu cez ktorý daná cesta vedie. Nakoniec je algoritmus ukončený na nejakom cieľovom vrchole a žiaden ďalší vrchol už nebude expandovaný. Po ukončení algoritmu je možné zostaviť najmenšiu cestu z počiatočného vrcholu k cieľovému vrcholu, a to jednoduchým spätným prechádzaním jednotlivých odkazov na rodičovské vrcholy.

Aby bol algoritmus čo najefektívnejší, mal by expandovať čo najmenší počet jednotlivých vrcholov. K dosiahnutiu toho je potrebné, aby bol neustále informovaný o tom, ktorý vrchol by mal byť expandovaný ako ďalší. V prípade, že sa expandujú vrcholy, ktoré vôbec nevedú k minimálnej ceste ide o zbytočné výpočty a mrhanie zdrojmi. Z toho je zrejmé, že taký algoritmus potrebuje k svojim výpočtom funkciu, ktorá z dostupných vrcholov na expandovanie vyberie taký, ktorý je najvhodnejším na nasledujúce spracovanie. Funkciu ktorú `A*` využíva k ohodnoteniu jednotlivých vrcholov je možné zapísať v tvare

$$f(n) = g(n) + h(n) \tag{4.1}$$

pričom jej podzložka $g(n)$ je zodpovedná za výpočet skutočnej ceny od počiatočného vrcholu k danému vrcholu n a heuristická funkcia $h(n)$ určuje cenu od vrcholu n ku koncovému (hľadanému) vrcholu. V prípade výpočtu heuristickej funkcie ide o odhadovanú hodnotu na základe poznatkov domény, nad ktorou algoritmus pracuje. Množstvo problémov, ktoré sú riešené hľadaním minimálnej cesty v grafe zároveň obsahujú nejakú dodatočnú informáciu o situácii respektíve kontexte, v ktorom sa daný problém odohráva. To nám umožňuje správne formulovať výpočet odhadovanej hodnoty. Z príkladu vyššie o výpočte najkratšej trasy medzi rôznymi dvoma mestami môže $h(n)$ určovať vzdušnú vzdialenosť medzi nimi. Nakoľko žiadna z ciest medzi dvoma mestami nemôže byť kratšia než ich vzdušná vzdialenosť, je zároveň táto hodnota spodným odhadom skutočnej ceny. V prípade, že by hodnota

heuristickej funkcie bola nulová, algoritmus vôbec nepozná, prípadne nevyužíva znalosť domény nad ktorou pracuje a v konečnom dôsledu bude pracovať rovnako ako Dijsktrova metóda popísaná v kapitole 4.2. Ak je však použitá heuristika dobrým spodným odhadom skutočnej ceny, algoritmus bude expandovať vrcholy len v okolí minimálnej cesty. Ak by bola presným odhadom skutočnej cesty, hodnota $f(n)$ sa pre jednotlivé vrcholy počas celého výpočtu nezmení a vrcholy, ktoré niesú na miniálnej ceste budú mať stále väčšie ohodnotenie. Tým pádom sa expandujú vrcholy len na minimálnej ceste. Naopak, ak hodnota $h(n)$ je väčšia ako skutočná cena, algoritmus **nezaručuje** vypočítanie minimálnej cesty. Je dôležité uvedomiť si, že heuristika zohráva skutočne dôležitú úlohu nielen z pohľadu dĺžky výpočtu, ale aj spôsobu akým algoritmus pracuje a jej správným zvolením dokážeme nájsť minimálnu cestu skutočne rýchlo. Ak je príliš nízka, minimálnu cestu síce vypočítame, no dĺžka výpočtu bude dĺhšia. Ak je však hodnota príliš vysoká, strácame garanciu nájdenia minimálnej cesty, no algoritmus bude fungovať rýchlo. Na zreteli treba mať aj fakt, že výpočet heuristickej funkcie prebieha pri expandovaní jednotlivých vrcholov a značne ovplivňuje celkovú dĺžku výpočtu algoritmu. Môže teda nastať situácia, že čas potrebný k výpočtu heuristiky v konečnom dôsledku prevýši čas, ktorý by bol potrebný k expandovaniu niekoľkých uzlov navyše. Netreba zabúdať aj na fakt, že obe zložky funkcie $f(n)$ vzorca 4.1 by pre správne fungovanie mali byť v rovnakom merítku. Ak je výsledok $g(n)$ v metroch, tak aj $h(n)$ by mala výsledok vrátiť v metroch. V opačnom prípade algoritmus nemusí zohľadňovať heuristiku vôbec, alebo ju bude zohľadňovať až príliš a nebude fungovať správne. Algoritmus 4 v pseudokóde zobrazuje princíp fungovania metódy A*.

Algoritmus 4: A*

Input: A weighted graph $G = (V, E)$, starting vertex $s \in V$ and goal vertex $goal \in V$. Default cost of each vertex is ∞.

```
1   Q = MinPriorityQueue()
2   s.cost = 0
3   s.parent = nil
4   Q.enqueue(0, s)
5
6   while not Q.isEmpty() do
7       current = Q.dequeue()          /* dequeue item with lowest priority */
8
9       if current = goal then
10          finish
11
12      forall adj in graph.adjacent(current) do
13          newCost = current.cost + calculateCost(current, adj)
14          if newCost < adj.cost then              /* path has better cost */
15              adj.cost = newCost
16              adj.parent = current
17              priority = newCost + calculateHeuristic(adj, goal)
18              Q.enqueue(priority, adj)
19      end
20  end
```

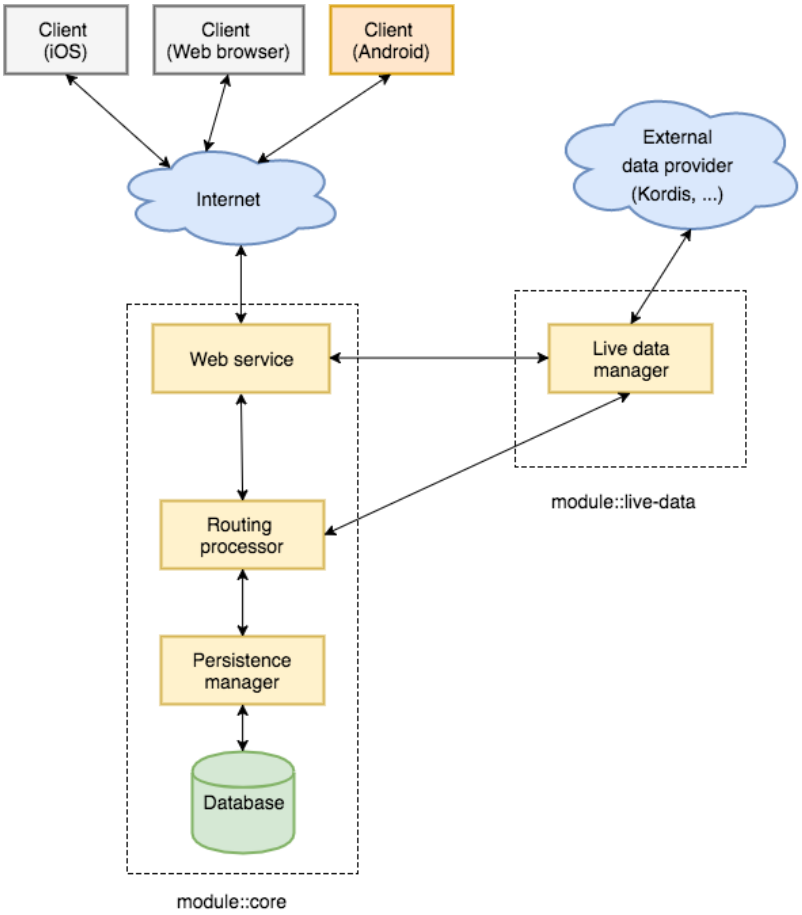
V inicializačnej časti algoritmus nastaví jednotlivé premenné počiatočného vrcholu a zaradí ho do fronty s prioritou 0. Následne cyklus na riadku `12` spracuváva jednotlivé vrcholy z fronty, až kým nenarazí na koncový vrchol alebo fronta nie je prázdna. Proces expandovania vrcholu pozostáva z nájdenia jeho bezprostredných následovníkov. Pre každého takéhoto následovníka sa vypočíta cena, s akou je potrebné dostať sa k danému vrcholu. V prípade, že je táto nová cena nižšia, než tá, ktorú má v danom čase vrchol uloženú, jej hodnota sa prepíše. Ak je podmienka vyhovujúca, taktiež sa upraví odkaz na ročičovský vrchol a vypočíta sa priorita, s ktorou bude daný vrchol vložený do fronty. Tá pozostáva zo súčtu ceny, s ktorou je potrebné dostať sa k tomuto vrcholu a z odhadu ceny (heuristiky), s ktorou sa vieme dostať z tohto vrcholu ku koncovému vrcholu. Nakoniec sa takýto následovník vloží do fronty s vypočítanou prioritou a cyklus sa opakuje.

4.4 Zhodnotenie

Každý z opísaných algoritmov má nepochybne svoje výhody, ale aj nevýhody. Či už sa to týka spôsobu akým algoritmus prehľadáva stavový priestor alebo pamäťovej, či časovej zložitosti. Kým Breadth First Search bol z pohľadu implementácie a princípu fungovania najjednoduchši, nesie so sebou aj značnú nevýhodu, a to tú, že nedokáže spracovať, respektíve nezohľadnuje ohodnotenie hrán v grafe. Dijkstrova metóda narozdiel od predošlého algoritmu toto obmedzenie nemá, je však z pohľadu implementácie a časovej/pamäťovej náročnosti zložitejšia, nakoľko využíva namiesto jednoduchej fronty prioritnú. Súčasne podlieha obmedzeniu ohodnotenia hrán, ktoré musí byť nezáporne. Posledný z opísaných algoritmov je A*, ktorý by sa dal nazvať vylepšenou verziou Dijkstrovej metódy. Výsledky a princíp jeho prehľadávania je do značnej miery možné ovplyvniť pomocou heuristickej funkcie, ktorá obrazne povedané, dokáže zmeniť ohodnotenie hrán a tým aj prioritu spracovania jednotlivých vrcholov čo sa v konečnom dôsledku premietne do celkovej dĺžky výpočtu.

Kapitola 5

Architektúra



Obr. 5.1: Architektúra serverovej časti aplikácie

Serverová časť aplikácie je zložená z dvoch hlavných modulov: `core` a `live-data`. `Live-data` modul slúži k získavaniu a spracovaniu reálnych dát z rôznych externých zdrojov. V prípade tejto diplomovej práce je takýmto externým zdrojom napríklad webová služba spoločnosti Kordis JMK, ktorá poskytuje informácie o vozidlách a odjazdoch z jednotlivých zastávok v rámci mestskej hromadnej dopravy v Brne. Tento modul prijíma a spracováva požiadavky od `Routing processoru` ako aj priamo od `Web service`. Modul `core`, do

ktorého patrí `Web service, Routing processor, Persistence manager`, zobrazený na obrázku 5.1, prijíma požiadavky od klientov prostredníctvom `Web service` a následne jednotlivé požiadavky spracováva. Spôsob komunikácie `Web service` s jednotlivými prvkami architektúry závisí od prijatej požiadavky:

- **plánovanie trasy** - v tomto prípade `Web service` komunikuje s `Routing processor`, ktorý vykoná všetky potrebné úkony k naplánovaniu požadovanej trasy (vrátane komunikácie s `Live data manager`, ak je to potrebné)
- **live dáta** - pri prijatí požiadavky, ktorého odpoveď má obsahovať súčasné údaje o vozidlách či odjazdoch `Web service` priamo kontaktuje `Live data manager`, ktorý mu požadované údaje poskytne
- **statické dáta** - ak požiadavka od klienta vyžaduje získanie statických dát, ktoré sú uložené v databáze, `Web service` komunikuje s `Persistence manager`, ktorý mu vráti požadované údaje a tie sú následne zaslané klientovi

Naviac, jeho úlohou je spracovávať statické dáta pridávané administrátorom ako sú napríklad cestovné poriadky či aktualizované informácie o zastávkach. Pre čo najväčšiu jednoduchosť je správa takýchto dát možná pomocou ovládacieho panelu popísaného v kapitole 6.3, ktorý je dostupný zo siete Internet na špecifickej URL[1].

[1] Uniform Resource Locator

Kapitola 6

Návrh serverovej časti

Táto kapitola je zameraná na návrh serverovej časti, ktorej hlavnou úlohou je plánovanie zvolenej trasy so špecifickými parametrami, ako aj spracovanie rôznych dát z externých zdrojov, a to najmä spracovanie údajov o vozidlách mestkej hromadnej dopravy mesta Brna. Tieto údaje sú poskytované v pseudo-reálnom čase. Pojem pseudo-reálny čas je použitý práve z dôvodu dostupnosti týchto dát, nakoľko obnova na strane dodávateľa prebieha len v určitých časových intervaloch. Naplánovanie zvolenej trasy by malo byť spracované v čo najkratšom čase, a to hlavne z dôvodu zvýšenia responzívnosti spolupracujúcich aplikácii. Súčasná doba poskytuje množsto rozličných technológií určených k rôznym potrebám. V tejto kapitole sú popísané jednotlivé zvolené technológie, dôvod ich zvolenia, ako aj k nim možné alternatívy. Sústredíme sa najmä na dátové úložiská, ktoré nepochybne zohrávajú dôležitú úlohu pri celkovej rýchlosti výpočtov. Ďalej je popísaná architektúra serverovej časti spolu s jej jednotlivými modulmi, dostupné aplikačné rozhranie, ako aj ovládací panel, ktorým je možné aktualizovať údaje v dátových úložiskách.

6.1 Databáza

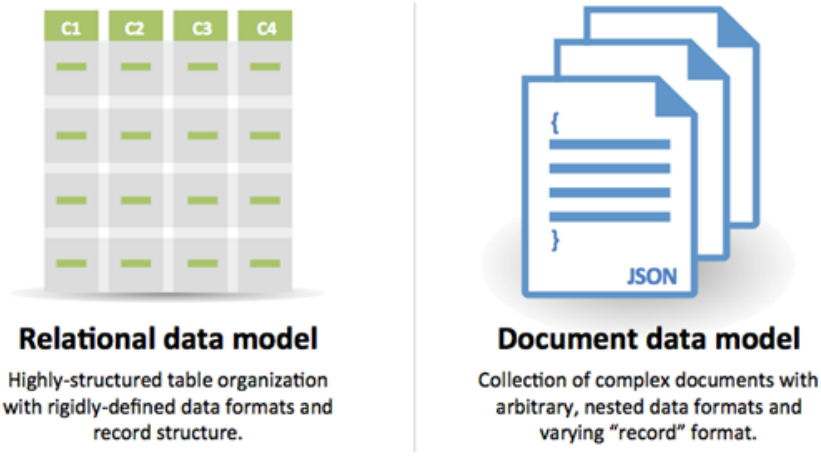
Tak ako v mnohých iných aplikáciach aj táto je závislá na perzistencii spracovaných dát. Takto uložené dáta môžu byť následne použité k výpočtom a získaniu požadovaných výsledkov. V súčasnosti existujú desiatky rôznych typov databáz a je teda potrebné zvážiť, ktorá z nich je pre danú aplikáciu tá správna. Prípadne, akú kombináciu je správne si zvoliť. Na zreteli treba mať nielen spôsob perzistencie, ale aj možnosti manipulácie s dátami aké nám daný systém riadenia ponúka.

6.1.1 SQL a NoSQL databázy

Vo svete databáz existuju 2 hlavné typy: SQL a NoSQL - relačné a nerelačné. Rozdiel spočíva primárne v tom ako sú vytvorené, aké dáta uchovávajú a akým spôsobom ich uchovávajú. Kým relačné databázy sú štrukturované, nerelačné sú tvorené vo forme dokumentov a môžu obsahovať dáta ľubovoľného, dynamicky meniaceho sa formátu.

SQL

SQL(`Structured Query Language`) je databázový programovací jazyk, používaný k správe dát v systémoch spravujúcich relačné databázy. SQL je štandardizovaný programovací jazyk vyvinutý spoločnosťou IBM pre získavanie, úpravu a vytváranie relačných databáz.[13]



Obr. 6.1: Porovnanie spôsobu ukládania dát v SQL a NoSQL databázach (Zdroj: dataconomy.com)

Systémy riadenia bázy dát (RDBMS) dnes tvoria výrazný podiel vo sfére informačných technológií a sú dominantnou technológiou pre perzistenciu dát bussiness a webových aplikácií. Aj napriek mnohým rozličným prístupom k ukladaniu dát či už ide o objektové databázy alebo XML[1] úložiská, ktoré sa udiali v posledných rokoch, nikdy nedosiahli na trhu a v používaní taký rozsah ako RDBMS.[25]

Spôsob ukládania dát v relačných databázach je pomerne striktný a takáto databáza pozostáva z jednej alebo viacerých tabuliek, ktoré obsahujú stĺpce a riadky. Tento spôsob je možné prirovnať k telefónnemu zoznamu. Každý riadok v takejto tabuľke reprezentuje jeden konkrétny záznam a každý stĺpec uchováva konkrétny typ informácie. Vzťah medzi jednotlivými typmi informácii, ktoré budú ukladané a tabuľkou sa nazýva **schéma**. Predtým, než je do tabuľky vložená akákoľvek informácia, schéma musí jasne definovať ako a aké dáta budú do tabuľky uložené. To so sebou prináša nevýhodu najmä v tom, že ukladané dáta musia mať jasnú štruktúru. Dobre navrhnutá schéma minimalizuje redundanciu dát a predchádza nesynchronizovanému stavu databázy. Na druhej strane zle navrhnutá schéma môže vyústiť do mnohých problémov, a to najmä z dôvodu rigidnosti. Ďalším z dôvodov obľúbenosti týchto databáz je, že sú súčasťou mnohých rozšírených programových „stackov“ ako napríklad LAMP(Linux, Apache, MySql, PHP). Medzi najpopulárnejšie SQL databázy a RDBMS patria:

- MySQL - najpopúlarnejšia open-source databáza, vhodná najmä pre CMS a blogy
- Oracle - objektovo-relačný systém riadenia bázy dát vytvorený v jazyku C++. Ide o platené, avšak veľmi spoľahlivé riešenie s rozšírenou zákaznickou podporou
- PostgreSQL - objektovo-relačný systém riadenia bázy dát, ktorý naviac od SQL používa procedurálne jazyky ako Perl a Python

[1] eXtensible Markup Language - https://www.w3schools.com/xml/default.asp

NoSQL

NoSQL databázy je možné v zjednodušenom slova zmysle chápať ako množinu súborov obsahjúcich jednotlivé dáta. Príkladom môže byť blogovací web, kde by každý súbor reprezentoval daný blog a obsahoval textovú časť, počet zdieľaní, komentáre, „like“ zo sociálnych sietí a podobne. Snaha uchovávať, spracovávať a analyzovať takéto dáta viedla k vytvoreniu databáz, ktoré nebudú obsahovať schému. Názov NoSQL je odvodený z anglického slova „Not only SQL“. Hlavnou výhodou NoSQL databáz oproti relačným je možnosť dynamicky meniť štruktúru uložených dát. To prispieva k väčšej flexibilite. Namiesto tabuliek sú NoSQL databázy dokumentovo-orientované. Týmto spôsobom môžu byť neštrukturované dáta ako fotky, videa, textové súbory, a pod. uložené jednotným spôsobom vo forme dokumentu. Tento spôsob je intuitívnejší, no vyžaduje si viac úsilia pri spracovaní a miesta pri ukladaní než štrukturované dáta v SQL databázach. Existuje niekoľko možných spôsobov, akým sú dáta v NoSQL databázach uložené:

- **Kľúč - hodnota** - najjednoduchšia forma ukládania dát do databázy. Dáta sú uložené vo forme indexovaných kľúčov s príslušnou hodnotou. Príkladom je Riak, Redis
- **Dokument** - komplexná dátová štruktúra nazývaná dokument, ktorá je spárovaná s unikátnym kľúčom. Dokument môže obsahovať rôzne dáta vo forme kľúč - hodnota, kľúč - pole alebo dokonca aj vnorené dokumenty. Dokumentovou databázou je napríklad MongoDB
- **Grafové databázy** - dáta sú uložené vo forme grafov; vhodné použitie napríklad pre sociálne siete. Databázy ukladajúce dáta vo forme grafov sú napríklad Neo4J alebo Giraph

Nasledujúca tabuľka zobrazuje porovnanie jednotlivých typov úložísk z pohľadu výkonu, flexibility, škálovateľnosti a komplexnosti.

Tabuľka 6.1: Porovnanie vlastností rôznych spôsobov ukládania dát v NoSQL databázach

Typ	Výkon	Flexibilita	Škálovateľnosť	Komplexnosť
Kľúč - hodnota	veľký	veľká	veľká	žiadna
Dokument	veľký	veľká	variabilná	malá
Graf	variabilný	veľká	variabilná	veľká

6.1.2 MongoDB

MongoDB je open-source databázový systém, ktorý patrí do skupiny najpoužívanejších NoSQL databáz pričom dáta sú ukladané vo forme dokumentov. Systém je vyvíjaný v jazyku C++ a dokumenty sú vo formáte podobnému JSON[2], ktorý sa nazýva BSON[3]. Veľkosť takéhoto dokumentu je obmedzená na 16MB. Vývoj začal už v roku 2007 firmou 10gen, ktorá v roku 2013 zmenila svoj názov na MongoDB, Inc.[12] Nakoľko je MongoDB zo skupiny NoSQL databáz a kolekcie nemajú pevnú schému, prvky v jednotlivých dokumentoch sa môžu líšiť a štruktúra dát môže byť v čase ľubovoľne zmenená. Dáta vo formáte JSON môžu byť benefitom aj z pohľadu kompatibility v prípade integrácie s inou službou. Naviac tento

[2]JavaScript Object Notation - http://www.json.org
[3]Binary JSON - http://bsonspec.org

```
{
    name : Susane,
    age : 24,
    address : {
        line : Hw Road,
        city : Chicago,
        state : Illinois
    }
    phone : [19041242632, 190824342234]
}
```

Kód 6.1: Vzor dokumentu v MongoDB

spôsob uľahčuje mapovanie medzi databázou a doménovým objektom. Databázový systém sa pýši hlavne veľkým výkonom, vysokou dostupnosťou a automatickou škálovateľnosťou. Medzi jeho popredné funkcie patria:[19]

- **Ad-hoc dopytovanie**
- **Replica Set** - skupina MongoDB serverov, ktoré uchovávajú rovnaké dáta čím zvyšujú rendundanciu a dostupnosť v prípade zlyhania
- **Horizontálna škálovateľnosť** - MongoDB umožňuje horizontálnu škálovateľnosť pomocou metódy nazývanej `sharding`. Sharding je metóda pre distribúciu dát naprieč viacerými zariadeniami. Narozdiel od vertikálnej škálovateľnosti, ktorá spočíva v zvyšovaní výkonu jedného zariadenia, horizontálna škálovateľnosť umožňuje rozdeliť jednotlivé úlohy na množinu podúloh, pričom každé zo série zariadení spracúvava konkrétu podúlohu. To umožňuje potencionálne vyššiu výkonnosť než spracovanie všetkých úloh na jednom serveri
- **Dopytovanie na základe lokácie** - v prípade dopytovania na základe lokácie je MongoDB jednou z mála NoSQL databáz, ktoré poskytujú geografické funkcie
- **Podpora viacerých enginov** - MongoDB poskytuje podporu viacerých databázových enginov ako napríklad WiredTiger či MMAPv1. Naviac poskytuje aj API pomocou ktorého je možné pripojiť vlastný databázový engine

6.1.3 Zhodnotenie a návrh

Po dôkladnom zhodnotení jednotlivých databázových systémov bol zvolený systém zo skupiny NoSQL databáz - MongoDB, ktorý okrem iného dosahuje veľký výkon, jednoduchú škálovateľnosť, neobsahuje schémy a poskytuje prácu s geografickými dátami. K mapovaniu dát medzi doménovými objektami aplikácie a databázou bol použitý framework Hibernate, ktorý patrí medzi popredné nástroje pre mapovanie dát v jazyku Java. Ide však o verziu OGM(Object Graph Mapping), ktorá je narozdiel od ORM určená pre NoSQL databázy. Framework je stále vo vývoji, a počas implementácie boli nájdené 2 chyby, ktoré však boli po nahlásení úspešne odstránené.

Nasledujúci príklad - algoritmus 5 zobrazuje konfiguráciu frameworku Hibernate pre Java EE aplikácie prostredníctvom `JPA`. Konfiguračný súbor je uložený v zložke

`resources/META-INF` s názvom súboru `persistence.xml`. Ako napovedá samotná prípona, konfiguračný súbor je vytvorený v jazyku XML a obsahuje následovné elementy:

- **persistence-unit** - určuje perzistentnú jednotku ako celok, najmä však jej názov, pomocou ktorého je následne možné v Java kóde získať referenciu na túto jednotku
- **provider** - element určujúci persistence providera v rámci `persistence-unit`, implementujúceho JPA. Konkrétne - `javax.persistence.spi.PersistenceProvider`.
- **properties** - množina vlastností, ktoré bližšie špecifikujú perzistentnú jednotku a to najmä typom databázy, jej názvom, prihlasovacími údajmi alebo adresou URL, na ktorej daná databáza pracuje

Algoritmus 5: Konfigurácia Hibernate

```
1  <?xml version=“1.0“ encoding=“utf-8“?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi=“http://www.w3.org/2001/XMLSchema-instance“
4      xsi:schemaLocation=“http://java.sun.com/xml/ns/persistence
5      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd“
6      version=“2.0“>
7
8      <persistence-unit name=“LyraPU“ transaction-type=“RESOURCE_LOCAL“>
9          <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
10         <properties>
11             <property name=“hibernate.ogm.datastore.provider“ value=“mongodb“
                />
12             <property name=“hibernate.ogm.datastore.database“ value=“LyraDB“
                />
13             <property name=“hibernate.ogm.datastore.host“ value=“localhost“ />
14             <property name=“hibernate.ogm.datastore.create_database“
                value=“true“ />
15             <property name=“hibernate.ogm.datastore.username“ value=“db_user“
                />
16             <property name=“hibernate.ogm.datastore.password“
                value=“top_secret“ />
17         </properties>
18     </persistence-unit>
19 </persistence>
```

Modelovanie dát je dôležitým prvkom pri návrhu databázy bez ohľadu na to či sa jedná o SQL, alebo NoSQL databázu. Existujú stovky kníh, ktoré popisujú ako správne modelovať dáta pre relačné databázy. No NoSQL databázy poskytujú maximálnu flexibilitu v spôsobe, akým budú dáta uložené. Pri správnom spôsobe je možné predísť mnohým komplikáciam, no v opačnom prípade si "priťahujete slučku okolo krku". Návrh a spôsob ukladania dát teda závisel nielen od samotnej štruktúry dát, ale aj od spôsobu akým MongoDB uchováva a načítava dáta z disku. Nakoľko je obmedzenie veľkosti dokumentu pomerne veľké, bolo možné a vhodné uložiť väčšie množstvo bližšie súvisiach dát do jedného dokumentu. Treba však mať na pamäti aj minimálnu efektívnu veľkosť dokumentu. MongoDB organizuje súbory

do 4kB stránok a pri načítaní sa načíta do pamäti naraz jedna celá stránka. Daná stránka pritom obsahuje dáta, o ktoré bola databáza požiadaná. Mongo však neukláda jednotlivé dokumenty v špecifickom poradí, a teda daná stránka môže obsahovať aj iné dokumenty. Toto sa nemusí zdať ako problém pri malom počte požiadavkov, no v prípade veľkého množstva dopytovania sa je to obrovské mrhanie už načítaných dát. Dokumenty, ktorých veľkosť je vačšia než veľkosť stránky niesú problémom, nakoľko MongoDB garantuje, že dokument je uložený na disku súvisle, a teda jeho načítanie je jednoduché. Samozrejme, medzi 4kB a 16MB je menšia veľkosť lepšia. Avšak pri modelovaní je potrebné vyhýbať sa veľkému množstvu potenciálne malých dokumentov.[24]

Jednou z chýb, ktoré pri návrhu modelu dát vznikli bola napríklad tá, že dokument typu `Stop`, ktorý reprezentoval „stĺpik“ v rámci jednej zastávky obsahoval dáta o príjazdoch a odjazdoch jednotlivých vozidiel k danej `Stop` ako aj zoznam `Stop`-iek, ku ktorým sa od tohto stĺpika dalo pomocou MHD určitým spôsobom dostať. To so sebou prinášalo hneď niekoľko problémov, a to nielen pri aktualizácii časov odjazdov a príjazdov v databáze, ale najmä pri plánovaní trasy. K tomu aby bolo možné zistiť kedy určitý spoj dorazí na nasledujúcu zastávku bolo potrebné načítať dokument o nasledujúcej `Stop`. Ďalej bolo potrebné nájsť najbližší možný príjazd k tejto `Stop` cez spoj, ktorý vyrazil z predchadzajúcej `Stop`. Navyše, deň, pre ktorý daný záznam odjazdu a príjazdu existoval bol uvedený v samostatnej premennej a časy príjazdov a odjazdov relatívne k tomuto dňu taktiež v samostatných premenných. To malo za následok nielen zbytočne vyššiu náročnosť na úložisko ale aj komplikácie vyhľadávania trasy pri prechode medzi dvoma dňami.

Po zistení a odstránení jednotlivých chýb bol vytvorený model dát pozostávajúci z dvoch typov kolekcií. Prvou je kolekcia dokumentov typu `Platform`, ktoré reprezentujú jednotlivé zástavky ako celok a obsahujú v sebe „stĺpiky“ vo forme vnorených dokumentov. To umožňuje udržiavať všetky informácie o konkrétnej zastávke v jednom dokumente. Druhou z kolekcií je kolekcia dokumentov typu `Timetable`, ktoré reprezentujú informácie o konkrétnom "stĺpiku", a to zoznam zastávok na ktoré je možné sa dostať, ako aj časové údaje o jednotlivých odchodoch vozidiel spolu s príchodmi k nasledujúcej zastávke. Tým sa redukoval počet načítaných dokumentov pri vyhľadávani trasy, nakoľko cenu k nasledujúcej zastávke bolo možné určiť aj bez nutnosti získavania informácií o taktejto zastávke.

6.2 Webové aplikačné rozhranie

K tomu, aby mohli byť jednotlivé funkcie aplikácie používané aj z externých zariadení, napríklad z webového prehliadača alebo mobilného telefonú či tabletu, alebo z ľubovoľného iného zariadia musí aplikácia poskytovať určité aplikačné rozhranie (skr. API). Rovnako ako grafické rozhranie uľahčuje prácu používateľom, tak aplikačné rozhranie uľahčuje prácu iným vývojárom. Toto rozhranie môže byť vytvorené napríklad pre:

- **knižnice** - aplikačné rozhranie je častokrát spájané s pojmom knižnica. API avšak definuje to, čo je knižnica schopná vykonať a samotná knižnica je implementáciou tohto API. Jedno API môže mať viacero implementácii. Oddelenie implementácie od API umožňuje, že v niektorých prípadoch môže program implementovaný v jednom jazyku používať knižnicu implementovanú v inom jazyku. Príkladom môžu byť jazyky Scala a Java, kde oba tieto jazyky po komplilácii vytvárajú bytecode, a teda vývojári pracujúci s jazykom Scala môžu používať Java API.[22]

- **webové aplikácie** - webové API definujú rozhrania, cez ktoré prebieha komunikácia medzi systémom a aplikáciou, ktorá využíva jeho zdroje. V kontexte vývoja webových

aplikácií je takéto API najčastejšie definované ako množina HTTP[4] požiadavkov spolu s definíciou štrukúry odpovedí. Zaužívaný formát odpovedí na jednotlivé požiadavky je XML alebo JSON, no nie je to vždy pravidlom. Z historických dôvodov je webové API častokrát spájané so slovným spojením „webová služba“. Súčasný trend však čím ďalej, tým viac upúšťa od komplikovanejších webových služieb využívajúcich protokol SOAP[5] a sústreďuje sa na jednoduchšie a priamejšie webové rozhrania typu REST[6].[4]

- **operačné systémy** - API pre operačné systémy poskytuje rozhranie, pomocou ktorého môžu aplikácie komunikovať s operačným systémom. Príkladom môže byť `Windows API`, ktoré poskytujú ohromnú spätnú kompatibilitu, a teda aplikácie vyvýjané pre staršie verzie systému Windows fungujú aj na novších, v tzv. `Compatibility mode`.
- **databázové systémy**
- **a mnohé iné**

Účel však zostáva stále rovnaký a tým je odbremeniť vývojárov od danej implementácie a poskytnúť im len "prístupové body", ktoré danú činnosť vykonajú.

6.2.1 REST

K tomu, aby bolo možné aplikáciu používať aj iným spôsobom, než len internou komunikáciou medzi jednotlivými modulmi bolo potrebné zvoliť si aplikačné rozhranie, cez ktoré bude možné komunikovať aj vzdialene. Napríklad pomocou Internetu. Za týmto účelom bolo zvolené rozhranie typu REST. Približne pred 17-timi rokmi študent Kalifornskej Univerzity Roy Fielding prezentoval svoju dizertačnú prácu popisujúcu `Representational State Transfer (skr. REST)`. Aplikačné rozhranie REST je architektinický štýl navrhnutý najmä pre distribuované systémy, konkrétne však pre WWW[7]. Vo svojej podstate ide o jednoduchú architektúru založenú na týchto siedmich vlastnostiach:

- **Výkonnosť** - ako interakcie medzi jednotlivými komponentami ovplivňujú výkon
- **Škálovateľnost** - podpora veľkého množstva komponentov
- **Jednoduchosť** - je zamerané na správnom rozdelení zodpovednosti
- **Modifikovateľnost** - miera náročnosti pre aplikovanie zmien na jednotlivé komponenty
- **Viditeľnosť** - bezbarierová komunikácia medzi komponentami
- **Prenositeľnosť** - jednoduchosť v premiestňovaní služieb z jedného prostredia na druhé
- **Spoľahlivosť** - miera odolnosti voči zlyhaniu

Naviac je tento štýl popísaný nasledujúcimi šiestimi „obmedzeniami“ slúžiacimi ako akýsi štandard, ktorý ho definuje.[10]

[4] Hypertext Transfer Protocol
[5] Simple Object Access Protocol
[6] Representational State Transfer
[7] World Wide Web

- **Uniformné rozhranie** - definuje rozhranie medzi klientom a serverom. To zjednodušuje a rozdeľuje architektúru na menšie celky čím umožňuje nezávyslý vývoj jednotlivých častí.

- **Bezstavovosť** - je kľúčovým faktorom tohto typu aplikačného rozhrania. Bezstavovosť značí, že server si uchováva stav o danej požiadavke len v rámci požiadavky samotnej. Či už je to v URI, tele požiadavky, parametroch alebo hlavičke. Klient musí v každej požiadavke uviesť všetky potrebné informácie o jeho stave. Na základe tohto stavu potom musí byť server schopný danú požiadavku spracovať, a to aj bez znalosti akéhokoľvek svojho interného stavu. Po spracovaní je výsledok a príslušný stav (ak je to potrebné) vrátený späť klientovi vo forme odpovedi (angl. response), a to v tele, návratovej hodnote a hlavičke. Takýto prístup zvyšuje najmä spoľahlivosť a škálovateľnosť. Spoľahlivosť z dôvodu, že to zjednodušuje zotavenie z prípadnej chyby. Škálovateľnosť zasa preto, lebo si server nemusí uchovávať interný stav o jednotlivých požiadavkách a následne tento stav zohľadnovať pri iných. Týmto spôsobom môžu byť použité zdroje po spracovaní ihneď uvoľnené.

  Na druhej strane so sebou bezstavovosť prináša aj nevýhody napríklad vo forme zvýšenia využitia siete, nakoľko každá požiadavka musí obsahovať všetky potrebné údaje o stave, čo so sebou v niektorých prípadoch prináša zbytočnú duplicitu dát naprieč rovnakými požiadavkami. Taktisto je potrebné sa spoľahnúť na to, že jednotlivé požiadavky od klientov budú implementované sématicky správne.

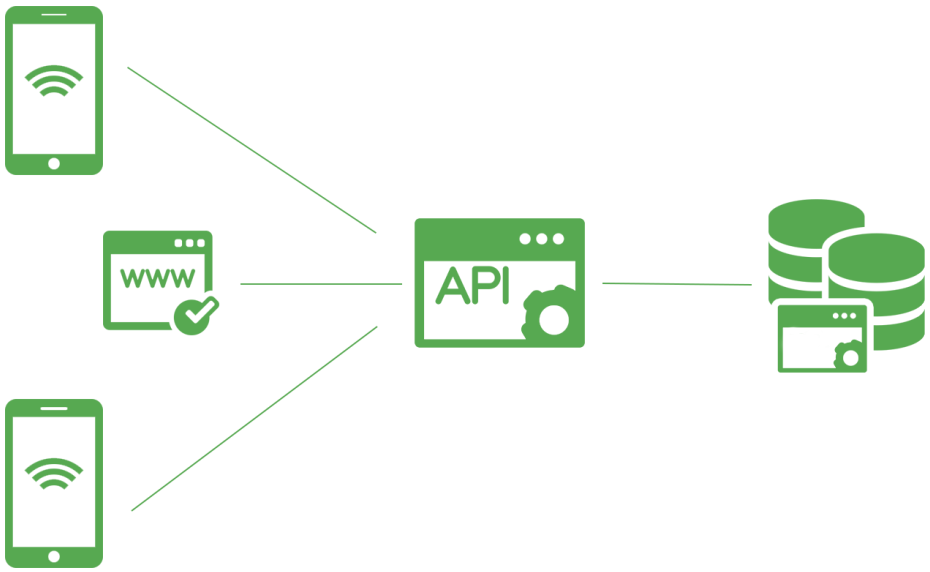
- **Cachovateľnosť** - umožňuje zvýšiť škálnovateľnosť a výkon služby. Odpovede musia implicitne alebo explicitne označeniť svoje dáta ako `cacheable` alebo `non-cacheable`. V prípade, že dáta budú označené ako `cacheable` to umožní klientovi pri rovnakej požiadavke na server znovu použiť dáta z predošlej požiadavky. V opačnom prípade budú ignorované. Týmto spôsobom je možné zredukovať počet reálnych požiadaviek na server, a tým zvýšiť nielen výkon služby a responzívnosť klienta, ale aj znížiť spotrebu dát. To môže byť veľmi užitočné hlavne pre mobilné zariadenia, ktorých dáta v prípade poskytovania operátorom sú stále finančne náročnou záležitosťou.

- **Klient-Server** - rozhranie oddeľuje úlohu klienta od úlohy servera. Klient sa napríklad nezaujíma o uchovávanie dát v databáze, čo je internou záležitosťou každého servera a tým sa zvyšuje možnosť prenosu. Na druhej strane, server sa zasa nezaujíma o grafické rozhranie, takže môže byť jednoduchší a viac škálovateľný. Klient aj server tak môžu byť vyvíjaný či nahradený nezávisle na sebe, avšak so zreteľom na neporušenie rozhrania medzi nimi.

- **Vrstevný systém** - umožnuje vytvoriť architektúru skladajúcu sa z viacerých vrstiev (modulov), čo zvyšuje škálovateľnosť, avšak za predpokladu, že každá z vrstiev vidí len na vrstvy bezprostredne okolo nej. Klient tak nedokáže určiť či sa spája s konečným serverom alebo len s „prechodným“. Prechodné servery resp. vrstvy môžu byť pridané za účelom zvýšenia bezpečnosti, oddelenia zastaralých klientov alebo služieb, cachovaniu a podobne. Na druhej strane hlavnou nevýhodou tejto vlastnosti je, že môže zvyšovať latenciu a záťaž pri spracovaní požiadaviek.

- **Kód na požiadanie (voliteľné)** - server umožňuje „odľahčiť“ klienta tak, že rozšíri alebo upraví jeho funkcionalitu tým, že presunie časť logiky, ktorú môže vykonať na seba. Príkladom môžu byť predkompilované komponenty ako Java applety alebo `JavaScript` skripty.

Jediným obmedzením definujúcim `REST`, ktoré je voliteľné, a to najmä z dôvodu znižovania viditeľnosti je *Kód na požiadanie*. Viditeľnosť umožnuje jednotlivým častiam architektúry monitorovať a regulovať interakciu medzi ďalšími časťami v rámci danej architektúry. Pri porušení akéhokoľvek iného z vyššie uvedených obmedzení nemožno považovať takúto architektúru za `REST`.



Obr. 6.2: Princíp fungovania REST API

REST kladie veľký dôraz na zdroje (angl. `resources`). Zdrojom je objekt, ktorý obsahuje dáta, pričom jednotlivé zdroje majú medzi sebou vzťahy a množinu operácií pomocou ktorých medzi sebou interagujú. Môže existovať aj kolekcia takýchto zdrojov, ktorá potom interaguje s inými zdrojmi alebo kolekciami.

Ďalším z dôvodov, ktorý robi REST architektúru jednoduchú na používanie je, že pre komunikáciu využíva HTTP protokol. Tento protokol okrem iného definuje aj množinu metód, ktorých cieľom je bližšie špecifikovať požadovanú akciu nad určitým zdrojom. Zdrojom môže byť súbor alebo výstup nejakej služby vykonávanej na strane servera. Vo svojom štandarde `HTTP/1.0`[5] boli definované tri základné metódy: `GET`, `POST` a `HEAD`. Približne o tri roky neskôr bolo v rozšírujúcom štandarde `HTTP/1.1`[9] pridaných ďalších 5 metód: `OPTIONS`, `PUT`, `DELETE`, `TRACE` a `CONNECT`. REST však využíva len tieto štyri:

- **GET** - metóda GET slúži k získavaniu dát, a to bez toho aby ich akýmkoľvek spôsobom upravovala. Táto metóda patrí do skupiny tzv. bezpečných metód (angl. `safe-methods`), ktorých cieľom je len získať informáciu a neupravovať stav serveru. Výsledné dáta sú vrátené vo forme `JSON` alebo `XML` spolu s návratovou hodnotou určujúcou úspešnosť danej operácie.
  - 200 - OK
  - 400 - chýbná požiadavka
  - 404 - požadované dáta neboli nájdené

```
https://www.example.com/items/12345
```

- **POST** - je určená k vytváraniu nových zdrojov. Konkrétne podriadených zdrojov voči nejakému nadriadenému. V konečnom dôsledku je potom táto požiadavka smerovaná na nadriadený zdroj, ktorý ho vytvorí a asociuje ho so sebou. `POST` metóda nie je považovaná za bezpečnú, nakoľko je určená k modifikácii stavu servera. Nepatrí ani do skupiny tzv. idempotentných metód, pretože opakované vykonanie rovnakej požiadavky nevyústi v rovnaký výsledok ako keby bola daná požiadavka vykonaná len jediný raz. Po úspešnom vložení server vráti návratovú hodnotu `201 - Vytvorený` spolu s odkazom na novovytvorený zdroj v hlavičke pod položkou `Location`.

  `https://www.example.com/items`

- **PUT** - je spájaná s úpravou už existujúcich zdrojov. Avšak, je potrebné poznamenať, že `PUT` môže byť využitá aj pre vytváranie nových zdrojov, a to v prípade, že určovanie výslednej URI, na ktorej by mal byť daný zdroj dosiahnuteľný, je v réžii klienta. Rovnako ako `POST` ani `PUT` nepatrí do skupiny bezpečných metód, nakoľko upravuje zdroje servera, avšak patrí do skupiny idempotentných metód. V prípade úspešného vytvorenia nového zdroja je návratová hodnota zaslanáná serverom `201 - Vytvorený`, avšak odkaz na zdroj sa už v hlavičke neuvádza, keďže je klientovi známy. Ak je `PUT` použitá k úprave zdrojov a operácia bola úspešná, server vracia návratovú hodnotu `200 - OK`.

  `https://www.example.com/items/12345`

- **DELETE** - slúži ako už napovedá samotný názov k vymazávaniu zdrojov určených pomocou URI. Metóda nepatrí do skupiny bezpečných metód, no patrí do skupiny idempotentných. Opakovaný pokus o vymazanie už vymazaného zdroja vyústi v chybu na strane servera, ktorý vráti príslušnú návratovú hodnotu. V prípade úspešného vymazania server vráti návratový kód `200 - OK`.

  `https://www.example.com/files/5344`

Ďalšou alternatívou ponúkajúcou sa k vytvoreniu webovej služby je prostredníctvom `SOAP`. SOAP je narozdiel od REST, ktorý je „len“ akýmsi súhrom pravidiel a obmedzení, skutočným protokolom. To robí architektúru webovej služby používajúcej SOAP výrazne komplexnejšou. V zjednodušenom príklade by sme mohli vnímať REST ako pohľadnicu a SOAP ako obálku. Jednoduchšie a lacnejšie je odoslať pohľadnicu než obálku, a taktiež k prečítaniu obsahu obálky je potrebné vykonať niekoľko krokov naviac než je to pri pohľadnici. SOAP komunikuje prostredníctvom XML, ktorý výrazným spôsobom zvyšuje objem dát, a teda aj pamäťovú náročnosť či čas potrebný k spracovaniu správy. Nepochybne táto technológia disponuje aj mnohými výhodami. Avšak s ohľadom na rýchlosť spracovania a čo najmenšie množstvo prenášaných dát bolo od tejto možnosti upustené.

6.2.2 Jersey - RESTful Web Services in Java

Implementovať webovú službu či už typu REST alebo SOAP nemusí byť tak jednoduché ako sa na prvý pohľad môže zdať. Najmä, ak k dispozícii nieje potrebná sada nástrojov. Pre vývojarov aplikácií pracujúcich s programovacím jazykom Java bolo práve z tohto dôvodu vytvorené `JAX-RS API`, ktoré značným spôsobom uľahčuje vytváranie RESTových webových služieb. `Jersey`[8] je framework pre programovací jazyk Java, ktorý je referenčnou

[8] https://jersey.java.net

implementáciou `JAX-RS`. Navyše poskytuje svoje vlastné API a nástroje, ktorými ešte väčšmi uľahčuje prácu pri vytváraní webovej služby. Práve tento framework bol použitý pri vývoji webovej služby k tejto aplikácii. K stiahnutiu je dostupný na oficiálnych stránkach či už vo forme `.jar` súborov alebo ako `dependency` pre zostavovací nástroj `Maven`.

Pre vytvorenie webovej služby prijímajúcej požiadavky od externých aplikácií je v prvom rade potrebné zaregistrovať v aplikácii servlet container, a to v konfiguračnom súbore `webapp/WEB-INF/web.xml` v prvku `web-app`. Príklad takejto konfigurácie je uvedený v algoritme 6. Okrem samotného servletu je potrebné uviesť aj cestu k triedam, ktoré budú dané požiadavky spracovávať - teda reprezentovať jednotlivé prístupové body. Taktiež je potrebné špecifikovať mapovanie URL, na ktorých bude servlet aktívny. URL je príponou k bázovej adrese a v našom prípade je to `/api/*`. Výsledné URL bude napríklad: `https://www.lyra.cz/api/*`

Algoritmus 6: Konfigurácia REST API

```
1  <servlet>
2      <servlet-name>RESTController</servlet-name>
3      <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
4      <init-param>
5          <param-name>jersey.config.server.provider.packages</param-name>
6          <param-value>cz.vutbr.fit.lyra.core</param-value>
7      </init-param>
8      <load-on-startup>1</load-on-startup>
9  </servlet>
10 <servlet-mapping>
11     <servlet-name>RESTController</servlet-name>
12     <url-pattern>/api/*</url-pattern>
13 </servlet-mapping>
```

Hneď ako je servlet zaregistrovaný je potrebné vytvoriť jednotlivé prístupové body, ktoré v prostredí Javy tvoria jednoduché POJO[9] objekty doplnené anotáciami z `JAX-RS` API. K tomu aby bolo možné určiť na akej URL je daný prístupový bod dosiahnuteľný slúži anotácia `@Path` z balíka `javax.ws.rs`. Tá prijíma jeden parameter typu `String`, ktorý špecifikuje relatívnu cestu od cesty zadanej v časti mapovania. Táto cesta bola určená pri pridávaní servletu v predchadzajúcom bloku kódu. Takto vytvorenú a anotovanú triedu je následne potrebné doplniť metódami, ktoré budú reprezentovať jednotlivé HTTP metódy. K tomu slúžia rôzne anotácie poskytnuté `JAX-RS` API a dostupné sú rovnako ako predošlá anotácia `@Path` v balíku `javax.ws.rs`. V nasledujúcom príklade - algoritme 7 je znázornené vytvorenie metódy `GET` prijímajúcej jeden parameter a vracajúcej dáta vo formáte JSON. Každá z vytvorených metód môže navyše bližšie špecifikovať svoju URL a to použitím anotácie `@Path` rovnako ako v prípade označenia celej triedy.

6.2.3 Návrh API

Z vyššie uvedených dôvodov bolo k vytvoreniu API, ktoré bude umožňovať komunikáciu aplikácie s externými aplikáciami použitá architektúra REST. Aj napriek viacerým možným metódam, ktoré REST prostredníctvom protokolu `HTTP` ponúka k manipulácii so zdrojmi

[9] Plain Old Java Object

Algoritmus 7: Prístupový bod vytvorený pomocou anotácií z JAX_RS API

```
1  import javax.ws.rs.*;
2
3  @Path("/v1/routes")
4  public class RouteService {
5
6      @GET
7      @Produces(MediaType.APPLICATION_JSON)
8      public Response getRoute(@QueryParam("id") String id){
9          ...
10     }
11 }
```

na strane servera je API vytvorené výhradne len jednou z nich. Pretože účelom externých aplikácii nieje akýmkoľvek spôsobom dáta modifikovať ale ich len získavať, dostupná je iba metóda `GET`, ktorá okrem iného patrí do skupiny bezpečných metód. Každá z odpovedí na lubovoľnú požiadavku má pevne stanovenú štruktúru obsahujúcu informácie o čase spracovania, verzii služby a v neposlednom rade obsahuje aj samotný výsledok spracovania v atribute `result`. Každá z odpovedí je vo formáte `JSON`. Z dôvodu pomerne komplexných štruktúr reprezentujúcich odpovede na niektoré požiadavky sú tieto štruktúry zobrazené v prílohe B.

Externé aplikácie tak majú možnosť získavať rôzne typy dát prostredníctvom nasledujúcich prístupových bodov, ktorých cesta je relatívna k ceste určenej v sekcii nastavení servletu. Prístupové body, ktorých URL cesta obsahuje slovo `live` sú určené k získavaniu reálných údajov o vozidlách, meškaniach či iných vlastností MHD v Brne.

- **routes** - umožnuje naplánovať trasu medzi dvoma rôznymi zastávkami v rámci MHD v Brne. Požiadavka má ako povinné tak aj voliteľné parametre, a v prípade absencie niektorého z povinných parameterov vráti server klientovi status s hodnotou 400. Počiatočná zastávka môže byť určená buď formou unikátneho identifikátoru alebo zemepisnou šírkou a výškou vo forme GPS súradníc, nie však súčasne oboma. Ďalším z povinných parametrov je unikátny identifikátor cieľovej stanice a časový údaj v milisekundách (Unixový čas) určujúci počiatočný čas plánovania. Následne je možné pomocou nepovinných parametrov bližšie špecifikovať maximálny počet prestupov, minimálny čas na prestup v sekundách, počet výsledkov a v neposlednom rade povoliť alebo zamietnuť zohľadnovanie aktuálnych pozícii vozidel pri procese plánovania.

- **platforms** - slúží k získaniu všetkých dostupných platforiem (zastávok) pre mestskú hromadnú dopravu v Brne. V prípade, že pri vykonávaní požiadavky nenastane žiadna chyba, server vráti príslušné dáta v tele odpovede spolu s návratovou hodnotou `200 - OK`.

- **platforms/{id}** - podobne ako prístupový bod `platforms` získa informácie avšak o špecifickej platforme, ktorej unikátný idetentifikátor je nahradený za parameter `id` v URL. Server vráti príslušné dáta a návratovú hodnotu s ohľadom na výsledok spracovania požiadavky.

- **live/vehicles?{codes}** - je určený k získaniu informácií o špecifických vozidlách MHD v Brne. Tento prístupový bod patrí do skupiny `live` a teda dáta, ktoré sú vrátené v odpovedi požiadavky obsahujú skutočné hodnoty o daných vozidlách v reálnom čase. Jednolivé kódy vozidiel sú špecifikované v URL ako zoznam a nahradené za parameter `codes`, pričom každý ďalší kód je oddelený znakom &. Výsledná požiadavka potom môže mať napríklad takúto formu URL.

  `.../live/vehicles?547&364&410`

- **live/near_departures?...** - rovnako ako `live/vehicles` aj tento prístupový bod patrí do skupiny `live` a jeho výsledkom je získanie informácií o nasledujúcich odjazdoch jednotlivých vozidel v parametrami špecifikovanom okolí. S požiadavkou je potrebné zaslať 4 povinné parametre, a to zemepisnú šírku a výšku (`lat, lon`) pri ktorej sa budú odchody vozidiel vyhľadávať, priemer (`diameter`) určujúcim priemer kruhu vyhľadavaného okolia, ktorý je špecifikovaný v metroch a v neposlednom rade maximálny počet vrátených záznamov (`limit`). Pri správne vykonanej požiadavke server spolu s návratovou hodnotou určujúcou výsledok operácie vráti aj príslušné dáta v tele odpovedi.

6.3 Ovladací panel

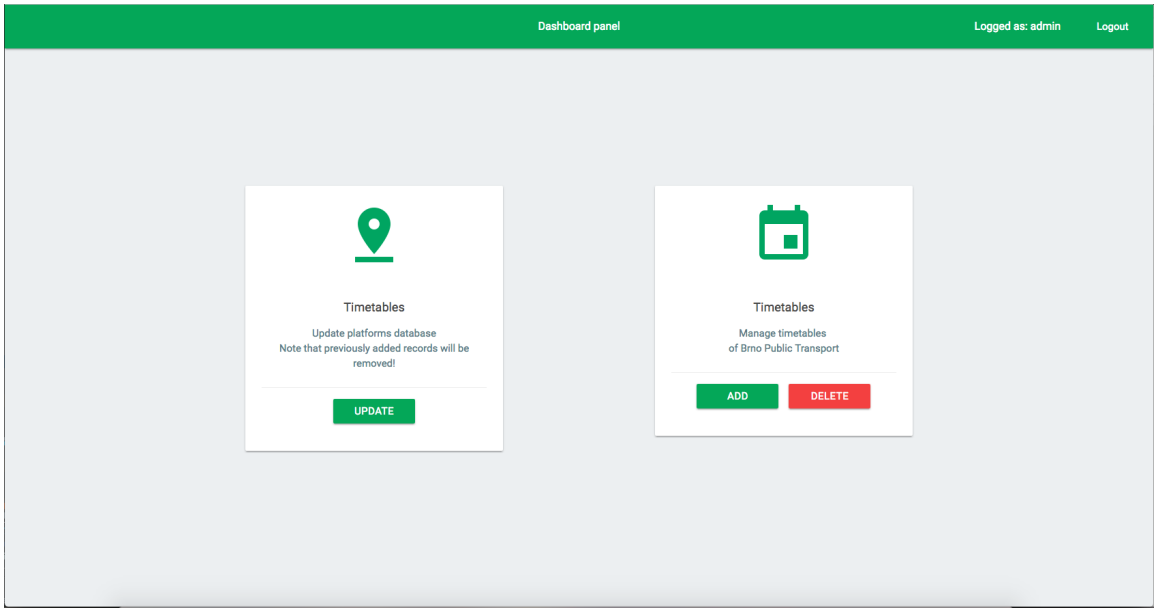
Jedným z cieľov tejto práce je vytvoriť aplikáciu na strane servera, ktorá bude pracovať s dátami týkajúcimi sa mestskej hromadnej dopravy v Brne. K tomu, aby bolo možné s týmito dátami manipulovať existuje niekoľko spôsobov. Jedným zo spôsobov je grafické rozhranie. Človeku je táto voľba asi najbližšia, nakoľko umožňuje zobraziť všetky možnosti na jednom mieste a zvýrazniť najdôležitejšie časti. Narozdiel od práce s konzolou, kde je častokrát k pochopeniu potrebné študovať mnohostránkové manuály je táto možnosť podstatne rýchlejšia. Avšak aj tu musia platiť isté zásady. Kvalitne vytvorené užívateľské rozhranie musí byť v prvom rade intuitívne, aby prácu uľahčovalo a nebolo naopak prítažou pri hľadaní jednotlivých možností.

Práve týmto spôsobom bol vytvorený ovládací panel k serverovej časti aplikácie, pomocou ktorého je možné spravovať dáta v databáze. Vytvorené rozhranie je jednoduché a minimalistické, no zároveň poskytuje všetky potrebné úkony k aktualizácii či pridávaniu dát potrebných pre proces plánovania trasy. Po spustení servera je dostupné na adrese.

`'contextPath'/dashboard/panel.xhtml`

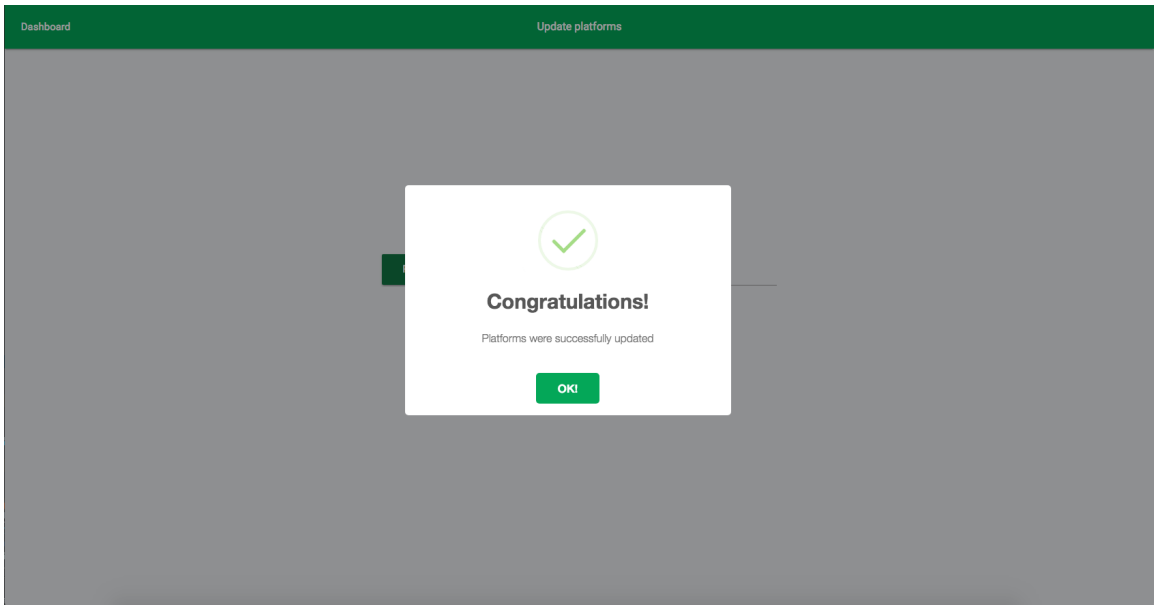
Pred samotným používaním je však potrebné sa do systému prihlásiť. Akákoľvek požiadavka smerovaná na sekciu vyžadujúcu prihlaseného užívateľa bude presmerovaná na prihlasovací formulár. Po úspešnom prihlasení je užívateľ presmerovaný na hlavnú obrazovku, ktorá zobrazuje práve prihlaseného užívateľa, možnosť odhlásiť sa, no najmä dve sekcie pre prácu so zastávkami a prácu s časmi odjazdov a príjazdov. Treba však poznamenať, že poradie pridávania či aktualizácie záznamov je dôležité, a to z dôvodu, že časy odjazdov a príjazdov sa vzťahujú ku konkrétnym zastávkam. Preto je potrebné ako prvé vložiť dáta popisujúce zastávky a následne je možné pridávať jednotlivé časové záznamy. Úvodná obrazovka je zobrazená na obrázku 6.3.

Po kliknutí na niektorú z možností sa zobrazí detailná obrazovka, kde je možné zvoliť si súbor zo systému a odoslať ho na spracovanie. To je indikované grafickým ukazovateľom



Obr. 6.3: Úvodná obrazovka ovládacieho panelu

a počas tohto procesu nemožno spracovávať žiadne iné súbory. Po úspešnom spracovaní sa zobrazí modálne okno oznamujúce výsledok operácie.



Obr. 6.4: Oznámenie o úspešnom spracovaní dát

6.3.1 JSF

Pre vytvorenie ovládacieho panelu bola zvolená technológia `JavaServer Faces`. JSF je MVC framework a štandardom pre tvorbu užívateľských rozhraní na strane serverov pomocou komponentov. Narozdiel od verzie `JSF 1.x`, ktorá k šablónovaniu používala `JavaServer`

Pages, verzia 2.x využíva pokročilejšie Facelets. Facelets sú oficiálnou technológiou pre vytváranie grafických prvkov (angl. views). Umožňujú definovať komponenty pomocou jazyka XML čím predchádzajú rôznym problémom, ktoré sprevádzali ich predchodcu JSP. Vytvorené grafické prvky zjednodušujú vývoj, znižujú veľkosť kódu, umožňujú prenositeľnosť medzi projektami. JSF je taktiež častokrát spájaná s podporou technológie AJAX, ktorá umožňuje vytváranie lepších, rýchlejších a responzívnejších webových aplikácií s podporou XML, HTML, CSS, JavaScript. Nové verzie JSF taktiež takmer úplne vyradili konfiguračný súbor faces-config.xml používaním anotácii. Ten okrem iného špecifikoval aj navigáciu medzi jednotlivými obrazovkami. Tú nahradilo jednoduché zadanie názvu požadovaného grafickeho prvku alebo Faceletu. JSF taktiež umožňuje aj prepojenie Faceletov s Java POJO objektami, a to vytvorením tzv. Managed Bean. K jej vytvoreniu je potrebné použiť anotáciu javax.faces.bean.ManagedBean vzťahujúcu sa na triedu. Trieda obsahuje klasické metódy getter & setter a business logiku. Ako bolo spomenuté vyššie, vďaka použitiu anotácie k vytvoreniu beany nebolo potrebné registrovať tento objekt v konfiguračnom súbore. Nasledujúci kód (algoritmy 8 a 9) zobrazuje porovnanie vytvárania Managed Bean pomocou konfiguračného súboru v JSF 1.2 a pomocou anotácie v JSF 2.x.

Algoritmus 8: Definovanie Managed Bean v JSF 1.2

```
1 <managed-bean>
2     <managed-bean-name>LoginBean</managed-bean-name>
3     <managed-bean-class>com.example.LoginBean</managed-bean-class>
4     <managed-bean-scope>session</managed-bean-scope>
5 </managed-bean>
```

Algoritmus 9: Definovanie Managed Bean v JSF 2.x

```
1 package com.example
2
3 import javax.faces.bean.ManagedBean
4 import javax.faces.bean.SessionScoped
5
6 @ManagedBean
7 @SessionScoped
8 public class LoginBean{
9     ...
10 }
```

Anotácia @ManagedBean naviac obsahuje dva voliteľné parametre. Prvým je name určujúci názov beany. Druhý, dôležitejší, určuje kedy má byť beana vytvorená. Jeho názov je eager a môže nadobúdať boolovskú hodnotu true alebo false. V prípade nastavenia na true bude beana vytvorená ešte predtým než je vôbec požadovaná. V opačnom prípade bude vytvorená až na požiadanie - tento spôsob je predvolený.

Životný cyklus beany je taktiež môžné definovať pomocou tzv. scopes. Tie určujú kedy bude inštancia beany zničená. V JSF 1.2 je to definované v konfiguračnom súbore pomocou elementu <managed-bean-scope> a v JSF 2.x zasa formou anotácií. Existuje šesť základných rozsahov, do ktorých môže byť bean-a zaradená.

- **@RequestScoped** - beana s týmto rozsahom pretrváva tak dlho, ako pretrváva HTTP požiadavka - odpoveď. Je vytvorená pri prijatí požiadavky a zničená hneď ako je odpoveď dokončená. Tento scope by mal byť použítý pre dáta, ktoré závysia na požiadavke. Napríklad dynamické zobrazovanie informácií na základe parametru pri metóde typu `GET`. V prípade, že beana nemá špecifikovaný scope, je použitý práve tento.

- **@NoneScoped** - takto označená beana pretrváva len počas vyhodnotenia jedného EL(Expression Language) príkazu. Vzniká pred vykonaním príkazu a zaniká ihneď po jeho vykonaní. JSF nikde neuchováva takto vytvorené beany, a teda v prípade zavolania rovnakého príkazu 5-krát, bude beana vytvorená a následne zničená rovnako 5-krát. Využitie tohto rozsahu je najmä pre dátové beany, ktoré budú pomocou `injection` vložené do iných bean.

- **@ViewScoped** - beana pretrváva kým užívateľ v okne/karte webového prehliadača interaguje s rovnakým View. Vzniká pri prijatí HTTP požiadavky a zaniká pri prechode na iné View. V prípade, že View je opustené požiadavkou typu GET, takto vytvorená beana nie je ihneď zničená, no nie je ani prístupná obvyklým spôsobom. JSF ju vloží do `UIViewRoot#getViewMap()`, kde kľúčom v mape je meno beany. Tento scope je vhodný pre komplexnejšie formuláre využívajúce AJAX, dátové tabuľky a komponenty využívajúce atribút `rendered`, ktorých stav musí byť uchovaný aj v nasledujúcich požiadavkách v rámci toho istého okna/karty(View) prehliadača.

- **@SessionScoped** - je určený pre beany, ktoré pretrvávajú počas celého HTTP spojenia (session[10]). Vzniká pri prijatí prvej požiadavky vrámci danej session, a zaniká hneď ako je session zneplatnená alebo je beana manuálne odstránená zo session mapy. JSF tieto beany ukladá do atribútu objektu typu `HttpSession`. Meno beany slúži ako kľúč v mape. Použitie je vhodné najmä pre dáta, ktoré môžu byť bezpečne zdieľané naprieč všetkými oknami vo webovom prehliadači v rámci jednej HTTP session. Ide napríklad o informácie o práve prihlásenom užívateľovi, zvolený jazyk, špecifické nastavenia užívateľa a podobne.

- **@ApplicationScoped** - takto označená beana zotrváva počas celého životného cyklu aplikácie. Je vytvorená pri prijatí prvej HTTP požiadavky (alebo ihneď po spustení aplikácie v prípade, že parameter `eager = true`) a zničená po ukončení aplikácie alebo ak je manuálne odstránená z application mapy. JSF tieto beany uchováva v atribute objektu `ServletContext` a meno beany slúži ako kľúč v mape. Takto vytvorená beana je vhodná hlavne pre dáta, ktoré môžu byť bezpečne zdieľané naprieč všetkými sessions - konštanty, statické dáta, nastavenia web aplikácie a podobne.

- **@CustomScoped** - umožňuje vytvoriť mapu (objekt typu `java.util.Map<K,V>`) a beana pretrváva dovtedy, kým je jej záznam prítomný v tejto mape. Mapu je potrebné vytvoriť manuálne v nejakom širšom scope, napríklad v `@SessionScoped`. Taktiež je potrebné dohliadnúť na prípadné odstránenie beany z mapy. Takýto scope je vhodný hlavne v prípadoch, kde žiaden iný z predvolených nedisponuje požadovanými vlastnosťami.

Pochopenie fungovania určovania rozsahu platnosti jednotlivých bean je veľmi dôležité pri tvorbe webovej aplikácie. Zlé určený rozsah môže spôsobiť, že prístup k dátam budú

[10] https://en.wikipedia.org/wiki/Session_(computer_science)

mať aj tí užívatelia, ktorí by ho za normálnych okolností mať nemali. Príkladom môže byť použitie `@ApplicationScoped` na miestach, kde by naopak mal byť použitý jeden z nasledujúcich scope: `session/view/request`. Okrem zneužitia dát sa môže výrazným spôsobom znížiť aj `UX(User experience)` a to v prípade použitia `@SessionScoped` na miestach, kde by mali byť použité `view/request` scopy. Dáta by tak boli zdieľané naprieč všetkými oknami/kartami v rámci jedného spojenia. To by mohlo spôsobovať nekonzistenciu pre užívateľa. Použitie `@RequestScoped` na beanu, ktorej dáta sú určené pre celé View by spôsobovalo znovu-inicializáciu premenných pri každom spracovaní (aj AJAX) požiadavky. To by mohlo vyústiť najmä v nefunkčné formuláre. V neposlednom rade môže zlé určený scope spôsobovať aj nežiadúci stav na strane servera, a to hlavne v prípade použitia `@ViewScoped` tam, kde by mohol byť použitý `@RequestScoped`. Klienta to síce nijako neovplyvní, no spôsobuje to zbytočnú pamäťovú záťaž na strane servera. Práve kvôli týmto príčinám je naozaj potrebné dôkladne zvážiť, aký scope je kedy a kde vhodný.

Pre správne fungovanie `JSF` je potrebné ich inicializovať. To sa robí rovnako ako v prípade `Jersey` pridaním minimálne servletu a mapovania do konfiguračného súboru `web.xml`, uloženého v zložke `webapp/WEB-INF`. Nasledujúci príklad v algoritme 10 zobrazuje základnú konfiguráciu `JSF`.

Algoritmus 10: Konfigurácia JSF

```
1  <servlet>
2      <servlet-name>Faces Servlet</servlet-name>
3      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
4      <load-on-startup>1</load-on-startup>
5  </servlet>
6  <servlet-mapping>
7      <servlet-name>Faces Servlet</servlet-name>
8      <url-pattern>/faces/*</url-pattern>
9  </servlet-mapping>
10 <servlet-mapping>
11     <servlet-name>Faces Servlet</servlet-name>
12     <url-pattern>*.jsf</url-pattern>
13 </servlet-mapping>
14 <servlet-mapping>
15     <servlet-name>Faces Servlet</servlet-name>
16     <url-pattern>*.faces</url-pattern>
17 </servlet-mapping>
18 <servlet-mapping>
19     <servlet-name>Faces Servlet</servlet-name>
20     <url-pattern>*.xhtml</url-pattern>
21 </servlet-mapping>
```

Kapitola 7

Návrh mobilnej aplikácie

Obsah tejto kapitoly je venovaný návrhu mobilnej aplikácie, ktorá je súčasťou zadania tejto diplomovej práce. Cieľom je vytvoriť mobilnú aplikáciu určenú pre mobilnú platformu Android[1], ktorá umožní užívateľom naplánovať požadovanú trasu v rámci systému mestskej hromadnej dopravy mesta Brna v Českej republike. Nakoľko je aplikácia určená pre široké spektrum užívateľov, zahrňujúce ako deti tak aj ľudí v pokročilom veku, jej vzhľad by mal byť minimalistický, no zároveň by mal reflektovať aktuálne trendy z oblasti návrhu užívateľských rozhraní. Intuitívne a rýchle ovládanie v aplikácii by malo byť taktiež samozrejmosťou. Jednotlivé požiadavky aplikácie sú doplnené diagramom prípadov použitia vytvoreného v jazyku UML [2] pre lepšie pochopenie kontextu. Medzi základné požiadavky, ktoré by aplikácia mala spĺňať patria:

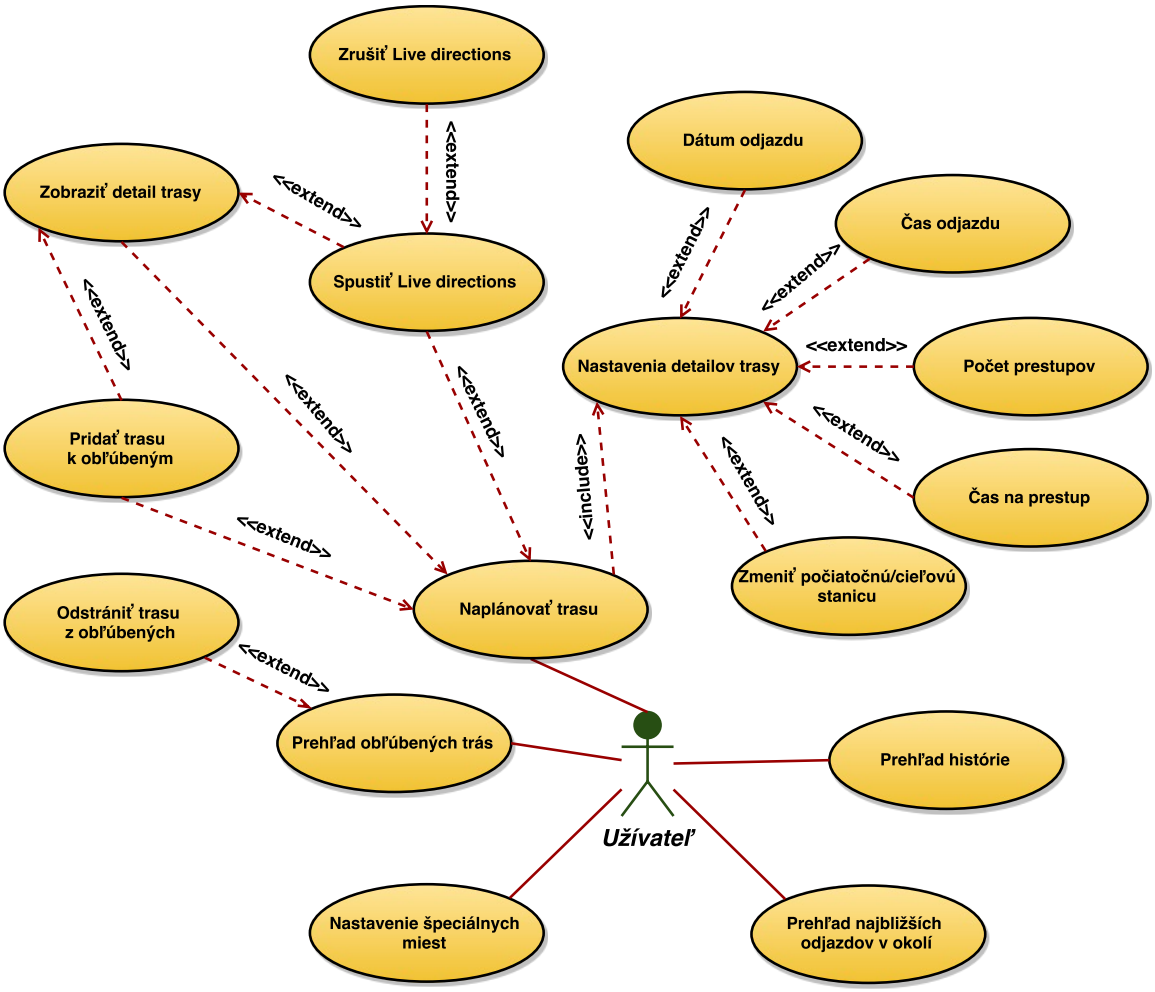
- **Plánovanie trasy** - umožniť užívateľovi naplánovať trasu medzi ľubovoľnými dvomi zastávkami v rámci MHD mesta Brna. Súčasťou procesu plánovania je aj možnosť bližšie špecifikovať požiadavky trasy
- **Uchovávanie dát** - možnosť uložiť špecifickú trasu do zoznamu obľúbených trás a nastaviť špeciálne miesta. Takto uložené dáta musia byť dostupné aj po opätovnom zapnutí aplikácie
- **Zobraziť detailný popis trasy** - vytvorenie detailného popisu trasy, ktorý zahŕňa čísla jednotlivých liniek, zoznam zastávok, časy príjazdov, odjazdov, miesta prestupov a podobne
- **Sledovať pohyb užívateľa pri cestovaní** - aplikácia by mala byť schopná sledovať pohyb užívateľa počas cestovania, zobrazovať mu aktuálne informácie o trase, prípadne ho upozorňovať na nadchadzajúce prestupy a meškania jednotlivých spojov. Táto funkcionalita je však úzko spätá s povoleniami, ktoré užívateľ aplikácií udelí.

7.1 Diagram prípadov použitia

Na obrázku 7.1 je zobrazený diagram prípadov použitia, vytvorený v jazyku UML, ktorý zachytáva jednotlivé možnosti interkacie užívateľa s aplikáciou. S aplikáciou pracuje len jediný typ užívateľa, ktorý má z hlavnej obrazovky prístup k takmer všetkým možnostiam

[1] https://www.android.com - internetová stránka mobilnej platformy Android
[2] Unified Modeling Language - http://www.uml.org

aplikácie spojených s procesom plánovania trasy. Jednak je to miesto k zadaniu koncovej zastávky a tým započatie procesu plánovania, ale aj rýchle plánovanie pomocou obľúbených miest, histórii či odchodov jednotlivých spojov v okolí.



Obr. 7.1: Diagram prípadov využitia mobilnej aplikácie

7.2 Požiadavky na užívateľa

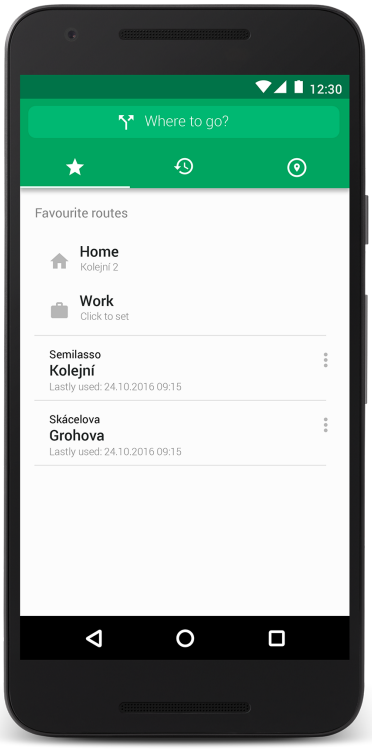
Ako pri každej mobilnej aplikácii, tak aj v tomto prípade rozhoduje užívateľ o tom, ako aplikácii dôveruje a aké povolenia jej k využitiu zdrojov udelí. Mobilná aplikácia vyvýjaná v rámci tejto diplomovej práce vyžaduje dve takéto hlavné povolenia. Prvým z nich je **pripojenie k internetu**, nakoľko plánovanie trasy nieje súčasťou implementácie v aplikácii, ale vykonáva sa na vzdialenom serveri. Ďalšou medzi požiadavkami je povolenie získavať **aktuálnu polohu** zariadenia, teda kde sa užívateľ v presnom časovom okamihu nachádza. Toto povolenie je **voliteľné**, a aplikácia bude fungovať aj bez neho. Avšak, pre správne fungovanie niektorých funkcií je nutné a bez neho k ním bude užívateľovi zamietnutý prístup.

7.3 Užívateľské rozhranie

Časť návrhu uživateľského rozhrania popisuje jednotlivé obrázky, ich účel, spôsob orientácie v rámci aplikácie na danej obrazovke, ale aj to ako sú medzi sebou jednotlivé obrazovky prepojené.

7.3.1 Úvodná obrazovka

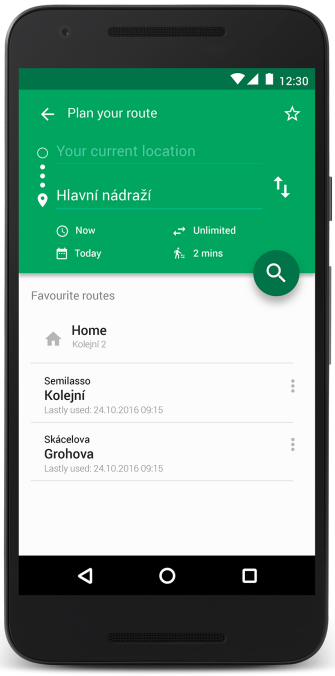
Hneď po spustení aplikácie, úplne prvou zo všetkých obrazoviek, ktoré sa užívateľovi naskytnú je hlavná obrazovka na obrázku 7.2. Tá ponúka užívateľovi hlavnú funkciu aplikácie, a tou je plánovanie trasy, ktorého proces sa začne hneď po vyplnení poľa v hornej časti obrazovky, do ktorého je potrebné vpísať cieľovú stanicu požadovanej trasy. Hneď pod spomínaným poľom sa nachádza panel na navigáciu, ktorý ako jednu z možností ponúka zobrazenie obľúbených trás alebo aj možnosť nastaviť si „špeciálne“ miesta pre rýchly prístup, ktoré sa nachádzajú vždy nad zoznamom obľúbených trás. Takéto miesta je možné spravovať po celú dobu užívania aplikácie. Súčasťou tejto obrazovky je však aj zobrazenie zoznamu histórie plánovania, ktoré je dostupné po kliknutí na strednú ikonu v navigačnom paneli. Poslednou časťou tejto obrazovky, nachádzajúcou sa pod tretím tlačidlom v navigačnom paneli je zobrazenie odchodov jednotlivých spojov v okolí, v ktorom sa užívateľ nachádza.



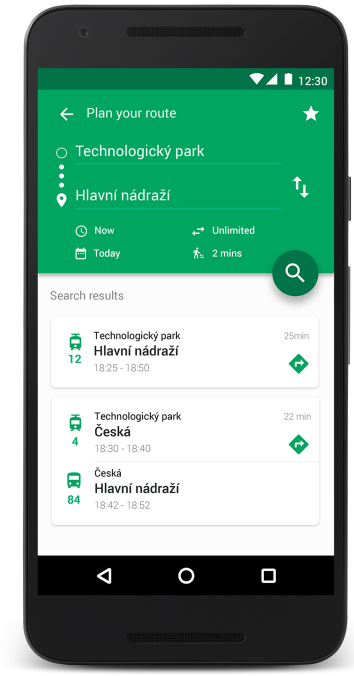
Obr. 7.2: Hlavná obrazovka mobilnej aplikácie

7.3.2 Plánovanie trasy

Nasledujúcim prvkom užívateľského grafického rozhrania, ktoré je dostupné po vpísaní koncovej zastávky do editačného poľa popisovaného v sekcii 7.3.1 je plánovanie trasy zobrazené na obrázku 7.3. Na tejto obrazovke má užívateľ možnosť presne špecifikovať jednotlivé parametre vyhľadávania, a to počiatočnú zastávku, cieľovú zastávku, deň a čas odkedy je možné plánovať trasu, ako aj počet prestupov či čas potrebný k jednotlivým prestupom špecifikovaný v minútach. Predtým, než užívateľ začne proces vyhľadávania jednotlivých trás sa mu v mieste pod nastaveniami ponúka zoznam obľúbených trás, a to za účelom zlepšenia UX[3] v prípade, že by sa užívateľ rozhodol použiť jednu zo svojich obľúbených trás. Avšak po tom, čo je vyhľadávanie ukončené a výsledky môžu byť zobrazené, nahradia sa nimi obľúbené trasy. Každý z výsledkov ponúka základný prehľad o danej trase akým je celková dĺžka, počet prestupov, použité linky či čas odjazdu ale aj čas príjazdu do cieľovej stanice. Taktiež je užívateľovi ponúknutá možnosť pri každom z výsledkov vyhľadávania zapnúť funkciu `Live directions` a to zeleným tlačidlom v hornej časti jednotlivých výsledkov vyhľadávania. Po kliknutí kdekoľvek inde na výsledok sa zobrazí detailný popis trasy, ktorý je opísaný v kapitole 7.3.3. Neoddeliteľnou súčasťou tejto obrazovky je aj možnosť uložiť si danú trasu medzi obľúbené, alebo ju odtiaľ vymazať.



(a) zoznam obľúbených trás zobrazený v oblasti výsledkov pred spustením výhľadávania



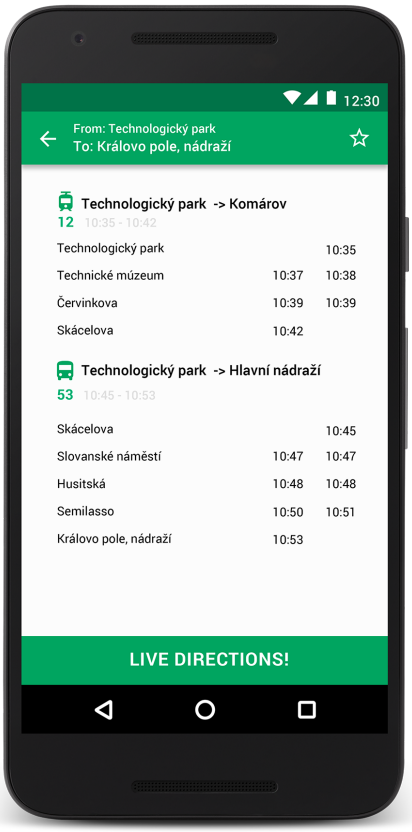
(b) výsledky zobrazené po vyhľadávaní

Obr. 7.3: Plánovanie trasy

[3]User Experience - https://en.wikipedia.org/wiki/User_experience_design

7.3.3 Detailný popis trasy

Hneď ako sa ukončí vyhľadávanie trás po zadaní jednotlivých nastavení zobrazených na obrázku 7.3 a kliknutí užívateľom na jeden z výsledkov vyhľadávanie, v aplikácií sa zobrazí detailný popis zvolenej trasy. Na tomto popise je možnosť videť kompletne naplánovanú trasu spolu s jednotlivými zastávkami, cez ktoré vedie, ako aj s časmi príjazdov alebo odjazdov z daných zastávok či číslami liniek, ktorými bude užívateľ cestovať. Nechýbajú ani možnosti ako pridanie alebo odobranie tejto trasy z obľubených položiek alebo spustenie funkcie `Live directions`. Detailný popis trasy je možné vidieť na obrázku 7.4.

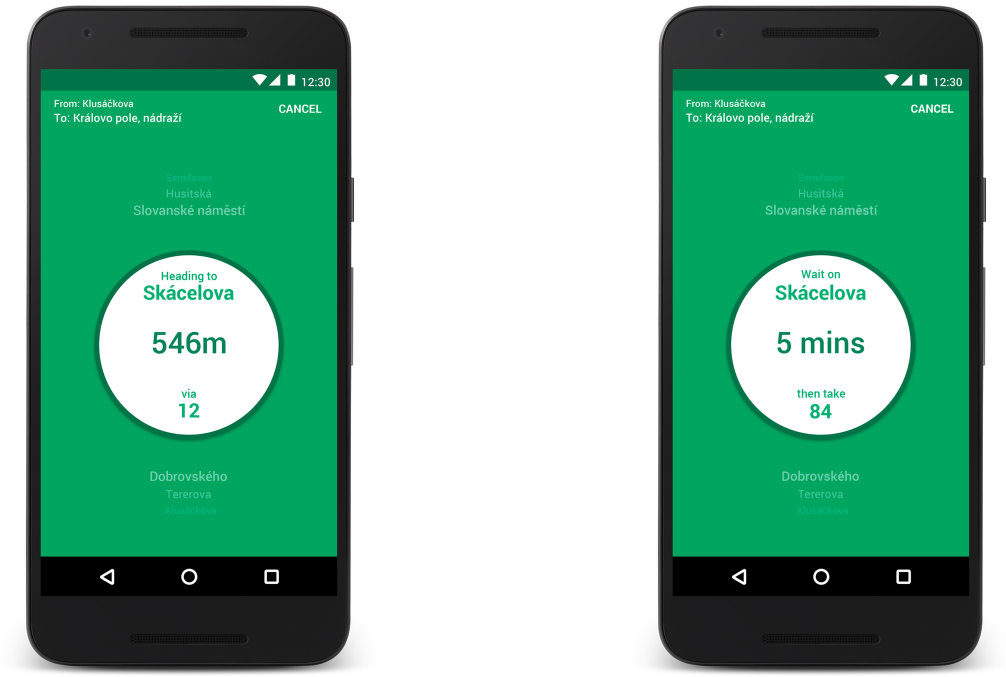


Obr. 7.4: Detailný popis trasy

7.3.4 Live directions

Záverečnou z obrazoviek a funkcií, ktoré sú v aplikácií dostupné je obrazovka 7.5 zobrazujúca funkciu `Live directions`. Navigácia k nej je možná či už priamo z výsledkov vyhľadávania konkrétnej trasy opísanej v kapitole 7.3.2 alebo z detailného popisu trasy, ktorý je vysvetlený v kapitole 7.3.3. V hlavnej časti tejto obrazovky sa nachádza kruhový indikátor, ktorý popisuje práve dejúcu sa akciu. Tou môže byť napríklad pohyb v MHD konkrétnym spojom, čakanie na prípoj či presunutie sa na inú zastávku. Jednotky, ktorými sa akcia vy-

jadruje sú závislé na typu akcie; v prípade pohybu sú to metre, no pri čakaní zasa minúty. Pod ním sa nachádza vertikálne zoradený zoznam posledných troch už prejdených zastávok. Naopak, nad ním je zasa zoznam nasledujúcich zastávok, taktiež zoradený vertikálne a limitovaný počtom tri. Úplne v hornej časti sa nachádza stručný popis trasy obsahujúci názov počiatočnej zastávky ako aj názov cieľovej zastávky. Napravo od nich je tlačidlo umožňujúce ukončenie tejto funkcie. V prípade, že práve dejúcou sa akciou je prestup medzi zastávkami je naviac v kruhovom indikátore zobrazená aj šípka smerujúca k vyžadovanej zastávke. Tá slúži ako kompas, nakoľko v niektorých pripadoch najmä pre cudzincov môže byť nájdenie toho správneho stĺpika v rámci danej zastávky pomerne náročné.



(a) pohyb v spoji MHD

(b) čakanie na prestup

Obr. 7.5: Plánovanie trasy

Kapitola 8

Implementácia

Táto kapitola je venovaná implementácii hlavných prvkov technickej časti práce. V úvodnej časti opisuje technológie použité pri vytvorení mobilnej aplikácie spolu so spôsobmi spracovania údajov na strane klienta. V druhej časti sa zameriava na serverovú časť, konkrétne na spôsob akým sú jednotlivé požiadavky zo strany klienta spracované a ako prebieha proces plánovania trasy od úplneho počiatku až po vytváranie odpovedí. Popisuje taktiež spôsob získavania dát zo služieb spoločnosti Kordis.

8.1 Android aplikácia

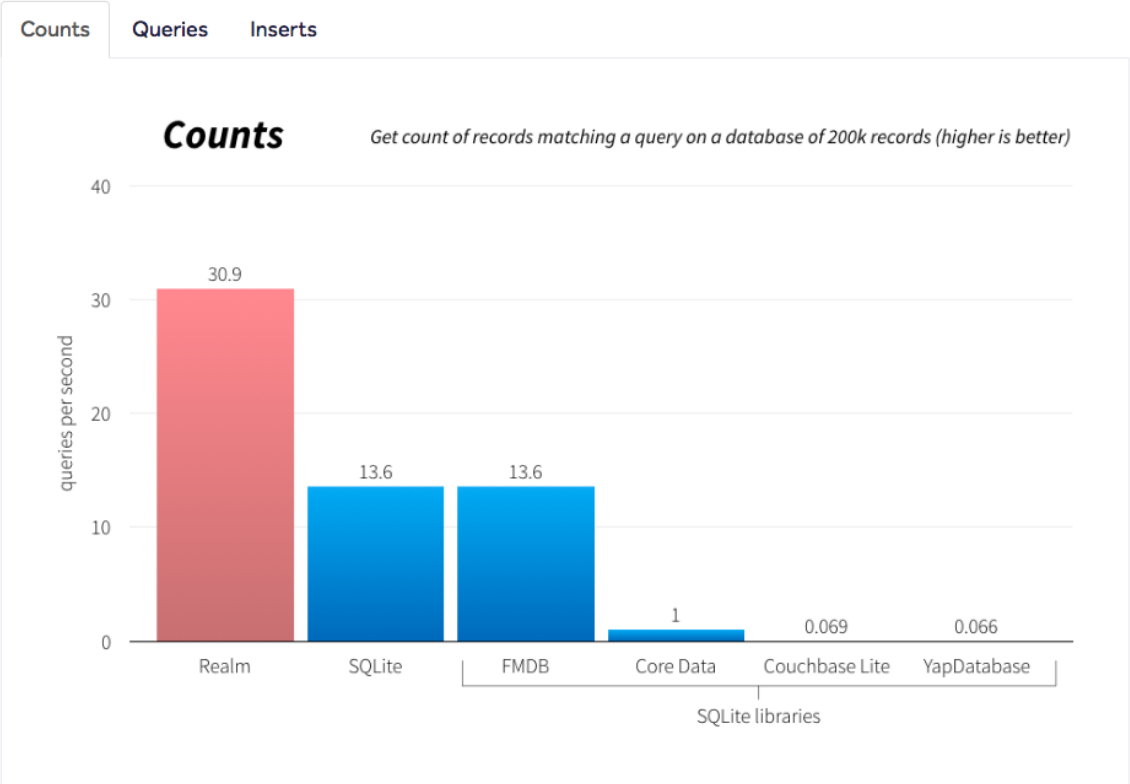
Jedným z cieľov tejto práce bolo implementovať mobilnú aplikáciu pre zariadenia s operačným systémom Android. Android open-source projekt vyvíjaný spoločnosťou Google, vytvorený na jadre Linuxu a je určený najmä pre mobilné zariadenia. Užívateľské rozhranie je zamerané hlavne na priamu interakciu s užívateľom pomocou dotykovej obrazovky a podporuje gestá, ktoré korešpondujú činnostiam v skutočnom svete. Od vydania prvej verzie tohto systému už uplynulo deväť rokov[21] a najaktuálnejšia verzia `7.1.2` s kódovým názvom `Nougat` bola vydaná v Apríli tohto roku[2]. Tá však podporuje len malý rozsah zariadení, a preto bolo potrebné vytvoriť aplikáciu so spätnou kompatibilitou aj pre zariadenia, ktoré nedisponujú najnovšou verziou systému. Vo výsledku je aplikácia funkčná pre všetky zariadenia so systémom Android od verzie 4.0.3 Ice Cream Sandwich (`API 15`). Všetky aplikácie v Androide pracujú v izolovanom prostredí nazývanom `sandbox`, a teda nemajú prístup k niektorým zdrojom zariadenia. Výnimkou je len priame povolenie zo strany užívateľa a to buď pred inštaláciou alebo za behu aplikácie[1] - podľa verzie systému. K vytvoreniu aplikácie bolo použité vývojové prostredie `Android Studio`, `Android SDK` spolu s ďalšími knižnicami a zostavovací nástroj `Gradle`.

8.1.1 Databáza

K ukládaniu dát na mobilnom zariadení bola použitá technológia s názvom `Realm`[2]. Tá narozdiel od SQLite, ktorá je základnym typom databázy pre mobilné aplikácie nevyžaduje vytváranie schém/tabuliek. Oboje z technológií podporujú `ACID`[3] transakcie. Súčasťou Realm knižnice je aj určitá forma mapovania perzistentných dát na Java objekty čím zvyšuje

[1] https://developer.android.com/training/permissions/requesting.html
[2] https://realm.io
[3] Atomicity, Consistency, Isolation, Durability

rýchlosť vývoja a odbremeňuje vývojárov od spracovávania dát pomocou kurzorov. Pre prípadné úpravy už existujúcich modelov slúži mechanizmus tzv. `Migrations`. Tie umožnujú dynamicky zmeniť štruktúru objektov pridávaním, odoberaním či úpravou jednotlivých atributov. Navyše, Realm uľahčuje vytváranie vzťahov medzi objektami, nakoľko mapovanie objektov prebieha na pozadí, bez nutnosti spájania roznych typov dát programátorom. K vytvoreniu modelových tried typu Realm je potrebné aby daná trieda dedila triedu typu `RealmObject`. To všetko je možné aj pomocou knižníc využívajúcich SQLite databázy, avšak za veľkú cenu výkonu. Hlavnou prednosťou Realm oproti SQLite je rýchlosť. Kľúčovou vlastnosťou rýchlosti je využitie tzv. `Zero-copy Architecture`[23]. Nasledujúci obrázok 8.1 popisuje rozdiel v rýchlosti získavania dát pomocou technológií `Realm` a `SQLite` alebo použitím knižníc vytvorených nad `SQLite`.



Obr. 8.1: Porovnanie technológií Realm a SQLite (Zdroj: Realm.io)

8.1.2 Live directions

Ako bolo spomenuté v kapitole 7.3.4, funkcia `Live directions` k svojej funkčnosti vyžaduje od užívateľa povolenie k polohe zariadenia. Bez nej nie je možné určiť presnú polohu a následne navigovať užívateľa pomocou špecifických krokov. Po spustení tejto funkcie sa automaticky na základe aktuálneho času vyhľadá zastávka, na ktorú užívateľ smeruje. V prípade, že cesta ešte nezačala bude táto zastávka zhodná s počiatočnou. Ak sa táto funkcia spustí po tom čo daná cesta skončila (čas príjazdu k poslednej zastávke je menší než aktuálny čas), je užívateľovi zobrazená chybová hláška a následne je vrátený na predchadzajúcu obrazovku.

Po úspešnom zistení nasledujúcej zastávky sa inicializujú jednotlivé prvky užívateľského rozhrania - predchadzajúce a nasledujúce zastávky, súčasná akcia, hodnota premennej spojenej s akciou a podobne. Zároveň sa začne zisťovanie meškania vozidiel spojených s touto trasou, ktoré je vykonané každých 30 sekúnd. Tieto dáta sú získavané od serverovej aplikácie a pri každej odpovedi servera na túto požiadavku sú podľa meškaní vozidiel prepočítané prípadné prestupy. Akonáhle nastane situácia pri ktorej by užívateľ niektorý z nasledujúcich prestupov kvôli meškaniu nestíhal, je o tom informovaný pomocou `Push` notifikácií. Navyše si môže zvoliť preplánovanie trasy od „problémovej“ zastávky alebo pokračovať v súčasnej ceste. Ak takáto situácia nastane a pokračuje sa ďalej v ceste, v okamihu keď sa meškania vozidiel upravia do takej miery, že prestupy medzi zastávkami bude znovu možné realizovať je úživateľ rovnakým spôsobom informovaný o tejto skutočnosti.

Zároveň sa po počiatočnej inicializácii aktivuje modul získavania polohy zariadenia s periódou obnovy 500-800ms. Pri každom notifikovaní systémom o získanej polohe je vypočítaná vzdušná vzdialenosť zariadenia od nasledujúcej zastávky. Pomocou tejto vzdialenosti a toho či je nasledujúca zastávka prvou v rámci prestupu (vrátane počiatočnej zastávky) je určený typ akcie, ktorú užívateľ vykonáva. Tá môže nadobúdať jednu z troch hodnôt `TRAVELING, WAITING, WALKING` pričom základnou je `WALKING`. Zároveň sa však zohľadňuje či je nasledujúca zastávka „dosiahnutá“. Tento stav nadobúda v prípade, že sa k nej užívateľ priblíži na vzdialenosť menšiu než 50 metrov v prípade akcie typu `TRAVELING` a 17 metrov v prípade `WALKING`. Spôsob určovania akcie je vyjadrený nasledujúcim spôsobom.

- **nasledujúca zastávka je prestupnou** - ak užívateľ na nasledujúcej zastávke začína svoju cestu alebo čaká na prestup akcia je určená na základe toho aká je veľká vzdialenosť medzi ním a zastávkou. V prípade, že jeho vzdialenosť je vačšia než 17 metrov a zastávka ešte nebola dosiahnutá je zvolená akcia `WALKING`. Naopak, ak zastávku už dosiahol a jeho vzdialenosť od nej je menšia než 50 metrov zvolenou je akcia `WAITING`. Ak je zastávka dosiahnutá a vzdialenosť užívateľa väčšia než 50 metrov prechádza sa do akcie `TRAVELING`. V tomto prípade sa zároveň nastaví za nasledujúcu zastávku ďalšia v poradí.

- **nasledujúca zastávka nie je prestupnou** - v tomto prípade je aktuálna akcia užívateľa `TRAVELING` a v prípade, že nasledujúcu zastávku ešte nedosiahol v tejto akcii aj naďalej zostrváva. Ak však zastávku už dosiahol, a zároveň je táto zastávka poslednou v sérii zastávok prejdených rovnakým vozidlom (užívateľ potrebuje prestúpiť na iný spoj) je nasledujúca akcia určená na základe toho či presup bude vykonaný na rovnakom stĺpiku alebo nie. Ak áno, zvolenou akciou je `WAITING`, ak nie tak sa zvolí akcia `WALKING`. Ak je zastávka dosiahnutá, **nie je** poslednou v sérii a zároveň sa užívateľ vzdialil na vzdialenosť väčšiu než 17 metrov zvolenou akciou je - `TRAVELING` a za nasledujúcu zastávku sa nastaví ďalšia v poradí.

V kruhovom indikátore zobrazujúcom okrem nasledujúcej zastávky a práve vykonávanej akcie je taktiež užívateľovi zobrazená aj aktuálna hodnota spojená s akciou. Tá ho udržiava informovaného o súčasnom stave. Ak je práve vykonávanou akciou cestovanie (`TRAVELING`) doplňujúcou informáciou je jeho vzdialenosť od nasledujúcej zastávky. V prípade akcie čakania (`WAITING`) je touto hodnotou rozdiel súčasného času a času odjazdu, pričom jeho hodnota môže byť zobrazená v troch formátoch: `hh:mm` ak je doba čakania väčšia než 1 hodina, `mm:ss` ak je doba čakania väčšia než jedna minúta, no zároveň menšia než jedna hodina alebo `ss` ak je doba čakania menšia než jedna minúta. Zvláštnym prípadom je akcia kráčania (`WALKING`), kde sa okrem vzdialenosti užívateľa od nasledujúcej zastávky na pozadí

indikátoru zobrazuje aj polo-transparentná zelená šípka. Tá funguje ako kompas a pomáha užívateľovi s nasmerovaním k správnemu stĺpiku v rámci zastávky. Smerovanie šípky je určované akcelerometrom pomocou ktorého je získaný gravitačný vektor a geomagnetickým senzorom, ktorého výstupom je vektor intenzity magnetického poľa v okolí. Následne je pomocou týchto hodnôt a interných funkcií Android frameworku spočítaná matica otočenia a azimut - uhol, medzi serverným magnetickým pólom a osou Y na zariadení. Na základe polohy nasledujúcej zastávky a vypočítanom azimute sa prepočíta uhol smerovania medzi zastávkou a zariadením. Pre zníženie záťaže zariadenia sa šípka prekresľuje len v prípade, že rozdiel medzi predošlým smerovaním a súčasným je väčší než jeden stupeň.

8.2 Serverová aplikácia

8.2.1 Získavanie dát v reálnom čase

Ako už bolo spomenuté v časti 5 popisujúcej architektúru serverovej časti, modul `live-data` a jeho implementácia majú úlohu sprostredkuvávať dáta o stave vozidiel MHD, odjazdoch, a podobne. K tomu je využívaná webová služba poskytovaná spoločnosťou Kordis JMK. Tá je však pre túto prácu dostupná len z jednej IP[4] adresy pridelenej zariadeniu na adrese `pcuifs2.fit.vutbr.cz`, ktoré sa nachádza v sieti VUT[5]. K tomu bolo potrebné vytvoriť na danom zariadení účet dostupný cez `ssh` a následne vytvoriť dynamický tunnel. Ten slúžil k tunelovaniu požiadaviek zo serveru cez dané zariadenie na službu Kordisu. Tá je popísaná vo formáte `WSDL` a komunikuje prostredníctvom protokolu SOAP. K tomu, aby bolo možné takúto službu využívať v prostredí JVM[6] je potrebné vytvoriť náležité Java triedy. Jedným z možných spôsobov je použitie nástroja `wsdl2java`[7], ktorý automaticky na základe `WSDL` popisu vygeneruje adekvátne triedy pripravené k použitiu. Pre umožnenie prístupu k webovej službe Kordisu bol vytvorený tunnel príkazom

```
ssh -D PORT login@pcuifs2.fit.vutbr.cz
```

Následne bol do projektu integrovaný nástroj `wsdl2java` ako Maven plugin zobrazený v algoritme 11. Po tom, čo bolo možné pristúpiť k službám spoločnosti Kordis, bol stiahnutý popis webovej služby vo formáte `.wsdl` a Maven príkazom

```
mvn generate-sources -Dmaven.test.skip=true -DsocksProxyHost=
    localhost -DsocksProxyPort=PORT
```

vygenerované zdrojové súbory, ktoré umožňovali túto službu používať.

Súčasťou implementácie modulu je aj cachovanie dát získaných z jednotlivých prístupových bodov, ktoré slúží k rýchlejšej odozve na požiadavky, ako aj k zníženiu zaťaženia externých služieb. Spôsob a časová platnosť cachovania závysí na konkrétnych typoch dát.

8.2.2 Plánovanie trasy

Proces plánovania trasy začína prijatím požiadavky od klienta a následným spracovaním jednotlivých parametrov. V prípade, že niektorý z povinných parametrov nie je špecifikovaný alebo formát ľubovoľného z parametrov je chybný, užívateľovi je vrátená chybová

[4] Internet Protocol
[5] Vysoké učení technické v Brne
[6] Java Virtual Machine
[7] http://cxf.apache.org/docs/wsdl-to-java.html

Algoritmus 11: wsdl2java Maven plugin

```
<plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>${cxf.version}</version>
    <executions>
        <execution>
            <id>test-service</id>
            <phase>generate-sources</phase>
            <configuration>
                <sourceRoot>/target/generated/src/main/java</sourceRoot>
                <wsdlOptions>
                    <wsdlOption>
                        <wsdl>/src/main/resources/kordis.wsdl</wsdl>
                        <serviceName>KORDISService</serviceName>
                        <extraargs>
                            <extraarg>-client</extraarg>
                        </extraargs>
                    </wsdlOption>
                </wsdlOptions>
            </configuration>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

hláška s príšlušnou návratovou hodnotou. Po úspešnom spracovaní parametrov je započatý proces plánovania v `core` module. V balíku `graph` sa okrem implementácie konkrétneho grafového algoritmu nachádzajú aj pomocné triedy, ktoré obsahujú rozhranie cesty, predvolené implementácie výpočtu heuristiky medzi dvoma uzlami v grafe a predvolenú implementáciu ohodnotenia cesty medzi dvoma uzlami. Tento balík má nasledujúcu štruktúru, pričom pre zachovanie čo najväčšej zrozumiteľnosti budú balíky ohraničené v hranatých zátvorkách:

- **Node** - objekt predstavujúci jeden konkrétny uzol vo vytvorenom grafe. Obsahuje informácie o zastávke, stĺpiku, použitom vozidle k dosiahnutiu tohto uzlu, čas príjazdu a odjazdu, odkaz na rodičovský uzol, cenu a mnoho iných vlastností, ktoré sú potrebné pre korektné a čo najlepšie naplánovanie trasy. Osobitne dôležitým je odkaz na rodičovský uzol pomocou ktorého je možné po ukončení plánovania zostaviť komplétnu cestu od počiatočného uzlu ku koncovému.
- [**path**] - tento balík obsahuje všetky potrebné rozhrania, ktorých implementácie sú použité k špecifikácii výpočtov pri plánovaní trasy ako aj pre samotné vyhľadávanie. Jedným z hlavných rozhraní je `Path`, ktoré reprezentuje už nájdenú trasu. Obsahuje základné metódy ako overenie či cesta bola nájdená, získanie prvého uzlu cesty, posledného uzlu cesty, prechod nájdenou cestou, textovú reprezentáciu cesty a pod.

Ďalším z rozhraní je `PathFinder`. Ide o generické rozhranie, ktorého parametrom je rozhranie typu `Path` a deklaruje metódy k samotnému vyhľadávaniu trasy. Návratovou hodnotou týchto metód je práve špecifikovaný typ rozhrania/objektu v hlavičke tohto rozhrania. Pre jednotné používanie, každá implementácia grafového algoritmu musí implemtovať toto rozhranie. V neposlednom rade tento balík obsahuje, ako už bolo spomenuté, aj ďalšie rozhrania, ktoré dodatočne špecifikujú správanie algoritmu alebo základnú implementáciu rozhrania `Path`. Sú to najmä rozhrania k výpočtom ceny, heurisitiky, konfigurácie trasy a ďalších.

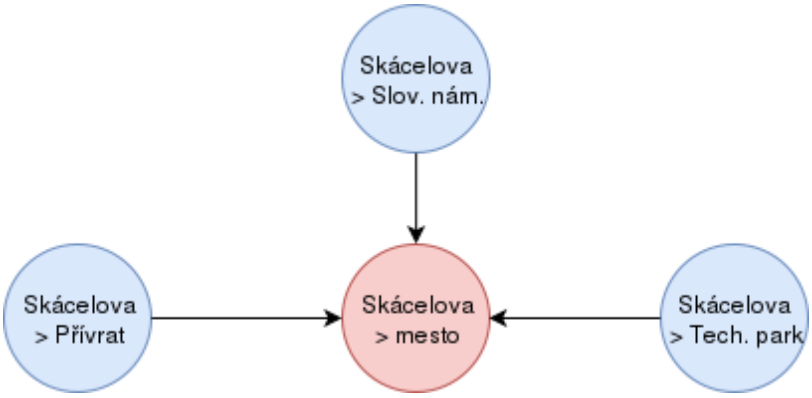
- **[evaluator]** - balík obsahujúci základné implementácie rozhraní, ktoré špecifikujú plánovanie a môžu byť použíté v rôzných algoritmoch
- **[algo]** - konkrétne implementácie jednotlivých algoritmov sú uložené v tomto balíku. Spolu s algoritmami obsahuje balík aj triedu používajúcu návrhový vzor `Factory`[8], ktorá slúži k získavaniu inštancií implementácií jednotlivých algoritmov.

K plánovaniu trasy bol implementovaný algoritmus `A*`, ktorý k svojim výpočtom používa aj implementácie rozhraní z balíka `path` opísaného vyššie. Tieto rozhrania sú použité práve k získavaniu ceny uzlov, výpočtu heuristiky, meškaní vozidiel alebo k samotnej konfigurácii plánovania a špecifikovať ich možno v konštruktore triedy. Následne sú uložené v inštančných premenných a v prípade potreby použité pri výpočte. Trieda má názov `AStar` a implementuje rozhranie `PathFinder` s použitým parametrom typu `Path`. Po prijatí a spracovaní požiadavky od klienta je teda vytvorená konkrétna inštancia triedy `AStar` a nad ňou zavolaná metóda implementovaná z rozhrania `PathFinder`, ktorá vráti naplánovanú trasu prostredníctvom inštancie implementujúcej rozhranie `Path`.

Na začiatku plánovania prebieha inicializácia pomocných premenných ako aj pridanie počiatočného uzlu do váhovo vyváženej fronty, z ktorej je pri vyberaní uprednostnený uzol s najnižším ohodnotením. Následne prebieha expanzia daného uzlu, a to až dovtedy, kým fronta nieje prázdna alebo uzol vybraný z fronty nieje koncovým uzlom - tzn. že zastávka, ktorú daný uzol reprezentuje je zhodná s koncovou zastávkou zadanou užívateľom.

Expanzia uzlu pozostáva z niekoľkých častí. Prvou časťou je nájdenie všetkých poduzlov, ktoré musia byť počas expanzie spracované. Je treba predpokladať, že užívateľ sa môže v rámci rovnakej zastávky presunúť medzi jednotlivými stĺpikmi, čo predstavuje nové potenciálne uzly grafu k spracovaniu. Každý uzol grafu okrem iného uchováva informáciu o zastávke a konkrétnom stĺpiku, ktorý reprezentuje. Pomocou toho je možné získať informácie o danej zastávke, a teda aj o všetkých jej stĺpikoch. Následne je pre každý stĺpik, ktorý nieje totožný s tým, ktorého uzol sa práve spracováva (aktuálne vybraný z frony) vytvorený poduzol. Poduzol zdieľa s uzlom, od ktorého bol vytvorený rovnakú zastávku, čas príjazdu, počet prestupov realizovaných po tento uzol a cenu cesty potrebnú k dosiahnutiu tohto uzla od počiatočného uzla. Čo však nezdieľa sú informácie o konkrétnom stĺpiku a použitom vozidle. Zároveň si poduzol uchováva referenciu na uzol, od ktorého bol vyvtvorený. Vzťah medzi rodičovským uzlom a poduzlom možno chápať ako prestup medzi vozidlami MHD vrámci jednej zastávky. Na obrázku 8.2 je možné vidieť vzťah rodičovského uzla označeného červenou farbou k poduzlom z neho vygenerovaných. Tie sú znázornené modrou farbou. Šípka smerujúca od poduzlov smerom k rodičovskému uzlu reprezentuje vzťah rodič - potomok. Keďže rodičovský uzol spolu s poduzlami majú rovnaké ohodnotenie, je možné a vhodné spracovať ich spolu.

[8] Factory pattern - https://www.tutorialspoint.com/design_pattern/factory_pattern.htm



Obr. 8.2: Vzťah rodičovského uzlu k poduzlom

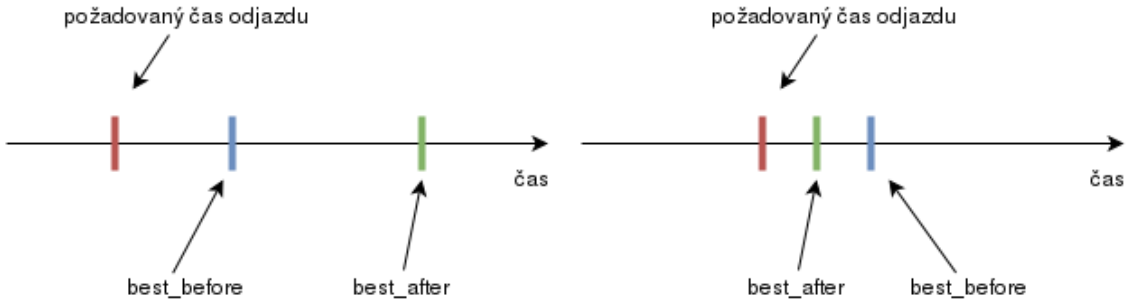
V druhej časti expanzie sú postupne po jednom spracovávané poduzly a rodičovský uzol. Pre každý z týchto uzlov je potrebné nájsť zastávky, ku ktorým sa z neho je možné dostať. Naviac, pre každú zastávku je nutné nájsť vhodný odjazd zo súčasnej pozície. Konkrétny najbližší čas odjazdu je stanovený podľa toho či ide o prestupný uzol alebo nie. Ak áno, k času príjazdu je naviac potrebné pripočítať čas potrebný na prestup. V opačnom prípade je najbližší čas odjazdu zhodný s časom odjazdu z cestovného poriadku. V oboch prípadoch však pri zohľadňovaní meškaní vozidiel treba k času príjazdu pripočítať aj aktuálne meškanie daného vozidla. Na základe toho či si užívateľ praje používať pri výpočtoch meškania vozidiel alebo nie je potom možné hľadanie najbližšieho odjazdu rozdeliť do dvoch kategórií:

- **meškania vozidiel niesú aktívne** - táto možnosť je pre výpočet jednoduchšia. Stačí, ak nájdeme podľa cestovného poriadku najbližší možný odjazd pre zadané vozidlo, zastávku a čas príjazdu

- **meškania vozidiel sú aktívne** - v tomto prípade je potrebné uvažovať niele situácie, kde hľadáme podľa cestovného poriadku najbližší odjazd od určitého času, ale aj situácie, kde príjazd nadväzujúceho vozidla na súčasnú zastávku je sice menší než najbližší požadovaný čas odjazdu, no meškanie tohto vozidla spôsobí, že sa príjazd oneskorí. V niektorých prípadoch až do takej mieri, že čas príjazdu presiahne najbližší požadovaný čas odjazdu - stane sa vyhovujúcim. Z tohto dôvodu prv je potrebné hľadať spoje, ktorých čas príjazdu k súčasnej zastávke je síce podľa cestovných poriadkov menší než najbližší požadovaný čas odjazdu, avšak spolu s meškaním spoja sa stane tento záznam vyhovujúcim. Takéto spätné vyhľadávanie prebieha dovtedy, kým čas príjazdu spolu s meškaním nebude menší než najbližší požadovaný čas odjazdu, pričom sa uchováva najviac vyhovujúci záznam. To je záznam, ktorého čas príjazdu spolu s meškaním vozdila čo najmenej presiahne najbližší požadovaný odjazd. Kvôli spätnej referencii, pomenujme tento záznam ako `best_before`.

  Aj napriek tomu, že sa nám podarilo záznam `best_before` nájsť, je stále potrebné nájsť aj najbližší možný odjazd podľa cestovného poriadku. Avšak, aj k takto nájdenému záznamu je potrebné pripočítavať meškanie vozidla, pomocou ktorého by bol realizovaný. Toto vyhľadávanie podľa cestovného poriadku s časom odjazdu **vačším** než je požadovaný sa opakuje dovtedy, kým sa nenájde záznam, ktorého čas odjazdu je vačší než čas odjazdu doteraz najviac vyhovujúceho záznamu - tzn. že sme dosiahli hranicu, kde aj v prípade že by meškanie vozidla príslušného záznamu bolo nulové,

bude takýto záznam stále horší. Najviac vyhovujúci záznam s časom odjazdu neskôr než je požadovaný označme ako `best_after`. Nakoniec sa zvolí medzi `best_before` a `best_after` ten, ktorý má výsledný čas odjazdu bližší k požadovanému. Obrázok 8.3 zobrazuje porovnanie rôznych pozícii `best_before` a `best_after` na časovej osi. V oboch prípadoch by bolo možné vďaka meškaniu vozidla stihnúť odjazd aj napriek tomu, že podľa cestovného poriadku už dané vozidlo odišlo (`best_before` je za požadovaným časom odjazdu). Na obrázku (a) je toto meškanie dostočne malé na to, aby nepresiahlo najbližší možný odjazd nájdený podľa cestovného poriadku, a teda `best_before` je uprednostnený pred `best_after`. Naopak, na obrázku (b) je meškanie príliš veľké, a to až tak, že presahuje najbližší možný odjazd iným vozidlom nájdeným podľa cestovného poriadku - `best_after` je uprednostnený.



(a) uprednostnený best_before (b) uprednostnený best_after

Obr. 8.3: Porovnanie výberu najbližšieho času odjazdu

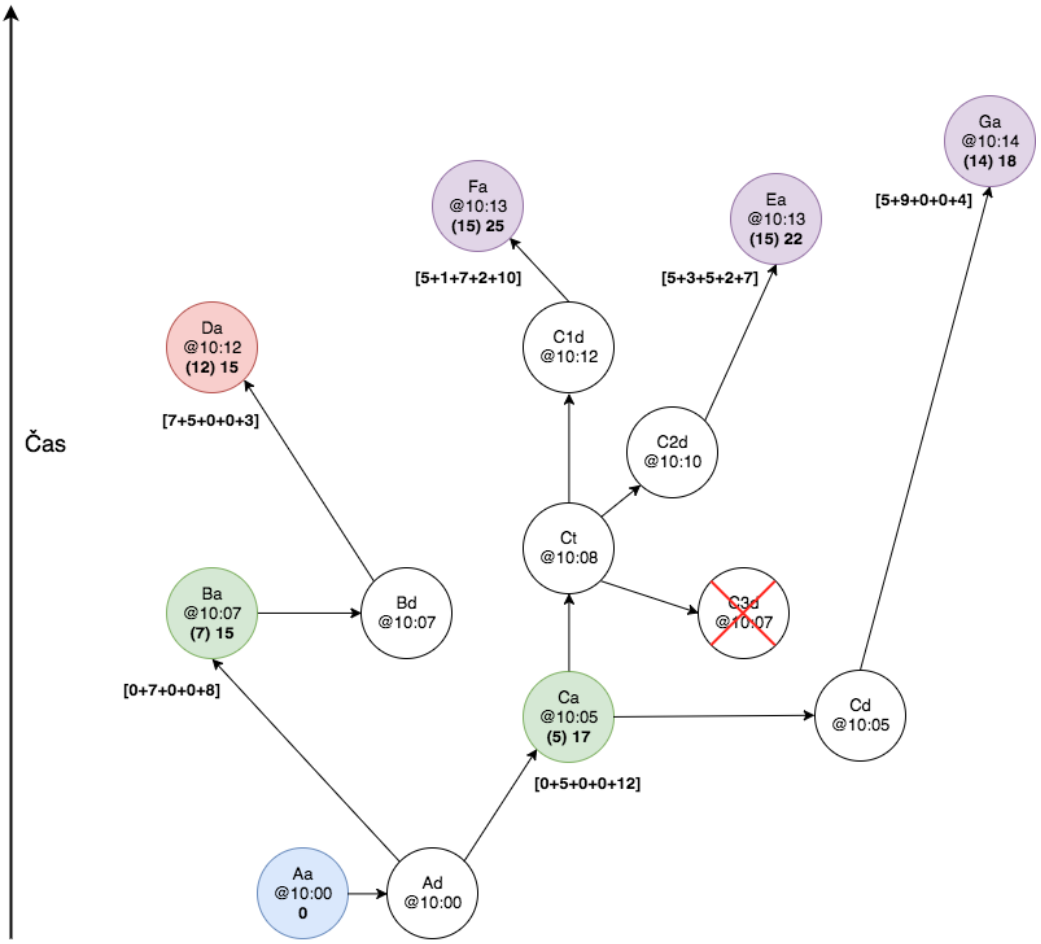
Po úspešnom nájdení najbližšieho odjazdu sa spočíta cena medzi súčasnou a nasledujúcou zastávkou. Tá je určená ako čas, ktorý je potrebný na presunutie sa medzi týmito dvoma zastávkami pomocou daného vozidla MHD. Inými slovami je to rozdiel času príjazdu a odjazdu medzi nasledujúcou a súčasnou zastávkou. Oba tieto časy sú podľa cestovného poriadku, nakoľko je potrebné zistiť len čistý čas. K výslednej cene je naviac pripočítaný aj čas čakania na zastávke ako aj časová penalizácia v prípade prestupu, ktorej hodnota je fixne určená na 120 sekúnd. Tá slúži najmä k znižovaniu počtu prestupov, ktoré nielen znižujú komfort cestovania ale aj zvyšujú riziko zmeškania niektorého z prípojov v prípade nečakaných udalostí. Príkladom môže byť nevhodné nájdenie cesty, ktorá je síce o 1 minútu rýchlejšia no vyžaduje si 1 prestup, pred cestou ktorá je priama - bez prestupov.

Výpočet ohodnotenia uzlu pri implementácii algoritmu A* si podľa vzorca 4.1 vyžaduje okrem ceny uzlu aj odhadovanú cenu k výslednému uzlu. Tá je vypočítaná ako vzdušná vzdialenosť medzi danými dvoma zastávkami. Následne je prevedená na časovú reprezentáciu, a to pomocou hrubého odhadu času potrebného k precestovaniu jedného kilometra vozidlom mestskej hromadnej dopravy. Ten bol zvolený na 180 sekúnd. K výpočtu vzdialenosti na guľovej ploche - ortodrome bola použitá Haversine formula[1], pomocou ktorej teda bolo možné vypočítať vzdušnú vzdialenosť medzi dvoma bodmi na Zemi. Po sčítaní oboch zložiek funkcie sme získali ohodnotenie uzlu, ktorý reprezentuje nasledujúcu zastávku. Výsledný vzorec pre výpočet ohodnotenia môžme definovať ako

$$v_{n+1} = c(n, s) + T + W + P + h(n + 1, f), \tag{8.1}$$

kde c je funkciou určujúcou cenu (ohodnotenie bez heuristiky) od počiatočného uzlu k súčasnému uzlu, T je čas, ktorý trvá samotná cesta vo vozidle, W je čas, ktorý vyjadruje čakanie

na prípoj, P je penalizácia (napríklad za prestup) a h je heuristická funkcia, ktorá odhaduje cenu potrebnú k prejdeniu z nasledujúcej zastávky do koncovej zastávky. Takto vytvorený a ohodnotený uzol je vložený do fronty len v prípade, že uzol reprezentujúci rovnakú zastávku sa vo fronte zatiaľ nenachádza alebo je jeho ohodnotenie horšie než ohodnotenie práve vytvoreného uzla. Po spracovaní rodičovského uzla a všetkých poduzlov je z fronty vybraný nasledujúci uzol a cyklus expanzie sa opakuje. Vytvorený podgraf výsledného grafu potom môže mať napríklad podobu zobrazenú na obrázku 8.4.



Obr. 8.4: Podgraf výsledného grafu pri hľadaní trasy

Čas je reprezentovaný na vertikálnej osi v smere zdola nahor. Kruhy predstavujú jednotlivé uzly v grafe, avšak do fronty su prídané len tie, ktoré sú zvýrazenené farebne. Naopak, nevyfarbené (biele) kruhy sú v grafe len za účelom lepšej vizuálnej reprezentácie popisujúcej expandovanie uzlov. Každý z uzlov obsahuje označenie, kde prvé písmeno v prvom riadku vyjadruje názov a druhé typ (`a` - príchod [arrival], `d` - odchod [departure], `t` - prestup [transport]). V druhom riadku sa nachádza časové razítko a v treťom riadku (okrem pomocných uzlov) je vyjadrená cena zapísaná v zátvorkách, ako aj celkové ohodnotenie. Pod každým z uzlov, ktorý je vložený do fronty sú taktiež zobrazené jednotlivé zložky výsledného ohodnotenia, a to v presnom poradí podľa vzorca 8.1. V prípade tohto podgrafu sú uzly do fronty vkladané v nasledujúcom poradí: modré, zelené, ružové, fialové.

Kapitola 9

Testovanie

Táto kapitola popisuje testovanie oboch častí práce no sústreďuje sa najmä na klientskú - Android aplikáciu. V prípade serverovej časti boli výkonané len výkonnostné testy, ktoré zobrazovali aký veľký vplyv má cachovanie jednotlivých údajov v pamäti oproti ich získavaniu priamo z databázy. Preukázalo sa, že v určitých prípadoch bol nárast výkonu až niekoľko desiatok tisíc percent. Testy boli spúšťané na počítači s procesorom Intel Core i5, 3.31GHz, 32GB DDR4 RAM a operačným systémom MacOS X. Konkrétne scenáre testovanie popisuje nasledujúca tabuľka 9.1:

Tabuľka 9.1: Porovnanie rýchlosti plánovania trasy

Počet zastávok na trase	Približný čas v milisekundách	
	Cache	Databáza
3	<5ms	180
6		250
9		350
15		750
20		1300
30		2000

Android aplikácia bola testovaná vybranou skupinou ľudí na zariadeniach LG Nexus 5 a One Plus Two ako z pohľadu užívateľského rozhrania, tak aj stability, presnosti detekcie zastávok či celkovej funkčnosti. Jedným z hlavných testovaných prvkov bolo užívateľské rozhranie, na ktoré sa kladol veľký dôraz. Muselo byť minimalistické a jednoduché, no zároveň poskytovať dostatočné množstvo prvkov na ovládanie aplikácie. Práve kvoli tomu prešiel návrh rozhrania niekoľkými fázami a upravované bolo najmä rozloženie prvkov a kombinácie farieb.

Okrem užívateľského rozhrania sa testovala aj stabilita v rôznych scenároch, ktorá odhlalila niektoré závažné chyby. Počas prvotnej fázy vývoja nebolo potrebné k testovaniu funkčnosti ísť do reálneho prostredia a k simulácii polohy postačovalo aj mockovanie súradníc priamo do GPS modulu zariadenia. To umožňovalo simulovať pohyb užívateľa bez toho, aby sa reálne nachádzal na niektorej zo zastávok alebo vo vozidle MHD. Akonáhle aplikácia spĺňala požiadavky v simulovanom prostredí, začali sa vykonávať testy v reálnom prostredí. Tie prebiehali v niekoľkých fázach pričom každá z fáz odhlalila pomerne závažné nedostatky. V reálnom prostredí bola testovaná takmer výhradne funčnosť časti `Live directions` a to na trasách:

- **Kolejní - Semilasso**
- **Kolejní - Hlavní nádraží**
- **Skácelova - Antoníska**
- **Husitská - Malinovského náměstí**
- **Česká - Komárov**

V nasledujúcich bodoch sú popísané fázy testovania tak ako nasledovali za sebou:

- **1. fáza (Problém priemernej hodnoty GPS súradníc)** - každá zo zastávok má na strane servera okrem presných GPS súradníc jednotlivých stĺpikov uloženú aj priemernú hodnotu týchto súradníc. Tá slúži slúži najmä k hrubému odhadu vzdialeností medzi jednotlivými zastávkami pri plánovaní trasy. Problém spočíval v tom, že aplikácia namiesto konkrétnych súradníc nasledujúceho stĺpika používala priemernú hodnotu čím sa navigácia v rámci `Live directions` stávala nepoužiteľnou.
- **2. fáza (Detekcia príchodu k zastávke)** - v druhej fáze testovania bolo odhalené, že vzdialenosť okolo nasledujúceho stĺpika, ktorá slúžila k detekcii príchodu užívateľa bola príliš malá. To spôsobovalo, že v prípade ak užívateľ cestoval dlhým vozidlom, a stál/sedel na jeho konci (väčšina vozidiel má GPS modul umiestnený v prednej časti), aplikácia nedetekovala príchod k zastávke aj napriek skutočnosti, že pri nej vozidlo zastavilo.
- **3. fáza (Pomalé obnovovanie polohy zariadenia)** - obnova lokácie zariadenia každé 4 sekundy spôsobovala problém v detekcii zastávok. Týkalo sa to výhradne zastávok na znamenie, pri ktorých šofér nemusí bez vyzvania cestujúceho zastaviť. Obnova polohy každé 4 sekundy pri rýchlosti približne 50km/h sposobí, že aplikácia vie porovnať aktuálnu polohu zariadenia s polohou a okolím zastávky približne každých 55 metrov. V niektorých prípadoch sa teda stávalo, že užívateľ prešiel okolo zastávky bez toho, aby ju aplikácia vôbec detekovala.
- **4. fáza (Chybné GPS súradnice zastávok)** - záverečná fáza testovania odhalila, že dáta poskytované treťou stranou, ktoré popisujú okrem iného aj polohy jednotlivých zastávok obsahujú chybné informácie. Aplikácii to spôsobuje čiastočné problémy pri navigovaní užívateľa, no po načítaní nasledujúcej zastávky sa aplikácia z chyby zotaví. Polohy boli overené aj pomocou Google Maps, ktoré taktiež poukazujú na chybné informácie. V testoch boli objavené dve takéto zastávky: Vědeckotechnický park, Technická. Nasledujúci obrázok 9.1 zobrazuje skutočnú pozíciu zastávky Technická(zobrazenú červeným kruhom s výplňou) voči pozícii, na ktorú poukazujú GPS súradnice (červený kruh bez výplne) získané z Google Maps a Kordisu.

Obr. 9.1: Porovnanie nekonzistencie polohy zastávky Technická

Kapitola 10

Možné rozšírenia

Mobilná aplikácia spolu so serverovou aplikáciou umožňujú kompletné naplánovanie trasy medzi dvoma rôznymi zastávkami v rámci hromadnej dopravy mesta Brno. Umožňuje ľubovoľne prispôsobiť proces plánovania, a tak čím viac vyhovieť požiadavkám užívateľa. Dá sa preto povedať, že funknčnosť oboch častí je kompletná. Existuje však niekoľko ďalších rozšírení, ktoré by mohli prispieť hlavne k spokojnosti užívateľov. Môžu k ním patriť napríklad tieto:

- **iOS aplikácia** - nakoľko sa táto diplomová práca z pohľadu tvorby klientskej aplikácie sústreďovala na platformu Android, bolo by vhodné rozšíriť klientskú časť aj pre zariadenia fungujúce na platforme iOS od spoločnosti Apple. Podporovať obe typy zariadení či už tablety alebo mobilné telefóny a zvýšiť tak počet úžívateľov, ktorí by túto aplikáciu mohli aktívne využívať.
- **Stiahnutie offline trás** - umožniť užívateľovi stiahnúť si naplánované trasy na niekoľko dní vopred. To by mohlo prispieť najmä k `user experience`, a to hlavne z dôvodu, že nie každý užívateľ má možnosť využívať dáta od operátora kdekoľvek a kedykoľvek. Väčšina tabletov dokonca nedisponuje možnosťou používať SIM kartu, a teda jediný možný spôsob pripojenia sa k Internetu je pomocou WIFI.
- **Android Wear** - podobne ako v prípade rozšírenia klientskej časti o aplikáciu pre platform iOS by bolo možné a vhodné rozšíriť funkčnosť pre zariadenia Google Wear. Tie umožňujú častokrát ešte rýchlejšie interagovať s užívateľom, nakoľko sú väčšinou priamo na ruke. V prípade notifikácie napríklad o meškaní vozidla by zariadenie mohlo upozorniť užívateľa zavibrovaním a následne zobrazením detailu meškania.
- **Podpora plánovania s presunmi medzi rôznymi zastávkami** - súčasná implementácia plánovania trasy umožňuje prestupovať na iné spoje, avšak len v rámci jednej zastávky. To čiastočne obmedzuje proces plánovania, keďže v niektorých prípadoch by presunutie sa na inú zástavku inak než pomocou MHD (napríklad pešo) mohlo urýchliť požadovanú trasu.

Kapitola 11

Záver

Cieľom tejto diplomovej práce bolo vytvoriť systém, ktorý bude schopný sledovať pohyb užívateľov v rámci mestskej hromadnej dopravy mesta Brna prostredníctvom mobilej aplikácie pre platformu Android, ale aj optimalizovať trasu a prestupy. Z tohto dôvodu bolo potrebné naštudovať si teóriu grafov, jednotlivé grafové algoritmy spolu s ich výhodami a nevýhodami, problémy spojené s grafmi, ako aj princípi spojené s návrhom užívateľských rozhraní či implementácou pre platformu Android. Výsledkom je aplikácia, ktorá okrem samotného naplánovania trasy umožňuje prostredníctvom sledovania polohy užívateľa upozorniť na prípadné zmeny pri prestupoch, ktoré môžu byť spôsobené najmä meškaním jednotlivých spojov. Táto funkcionalita je však podmienená povolením prístupu k polohe zariadenia zo strany užívateľa. Súčasťou výstupu serverovej časti je webové API, ktoré umožňuje komunikovať aj s inými než mobilnými klientmi, a to napríklad webovými prehliadačmi. Testovaním sa však ukázalo, že funkcia Live directions môže v ojedinelých prípadoch spôsobovať problémy s detekovaním zastávky, nakoľko sa skutočné polohy zastávok nezhodujú s údajmi poskytovanými treťou stranou.

Literatúra

[1] Abramowitz, M.; Stegun, I. A.: *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 10th printing*. Dover Publications, 1972, ISBN 978-0486612720.

[2] Amadeo, R.: *Android 7.1.2 leaves beta, arrives on Pixel and Nexus devices*. 2017, [Online; navštíveno 15.5.2017].
URL https://arstechnica.com/gadgets/2017/04/android-7-1-2-leaves-beta-arrives-on-pixel-and-nexus-devices/

[3] Bellman, R.: *Dynamic Programming*. Princeton University Press, 1957.

[4] Benslimane, D.; Dustdar, S.; Sheth, A. P.: *Services Mashups: The New Generation of Web Applications*. *IEEE Internet Computing*, ročník 12, č. 5, 2008: s. 13–15, [Online; navštíveno 3.5.2017].
URL http://corescholar.libraries.wright.edu/cgi/viewcontent.cgi?article=2126&context=knoesis

[5] Berners-Lee, T.; Fielding, R.; Frystyk, H.: *Hypertext Transfer Protocol – HTTP/1.0*. 1996, [Online; navštíveno 3.5.2017].
URL https://tools.ietf.org/html/rfc1945#section-8

[6] Bondy, J. A.; Murty, U. S. R.: *Graph Theory With Applications*. Elsevier Science Publishing Co., Inc., 1976, ISBN 0-444-19451-7.

[7] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to algorithms*. Cambridge, Mass: MIT Press, 2009, ISBN 978-0-262-03384-8.

[8] Dijkstra, E.: *A note on two problems in connexion with graphs, in: Numerische Mathematik 1*. Springer Berlin Heidelberg, 1959, p. 269–271.

[9] Fielding, R.; Gettys, J.; Frystyk, H.; aj.: *Hypertext Transfer Protocol – HTTP/1.1*. 1999, [Online; navštíveno 3.5.2017].
URL https://tools.ietf.org/html/rfc2616#section-9

[10] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, University of California, Irvine, 2000.
URL
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[11] František, Z.; ml., Z. F.: *Studijní opora: Základy umělé inteligence*. 2012, [Online; navštíveno 05.01.2017].
URL
https://www.fit.vutbr.cz/study/courses/IZU/private/oporaizu-esf-5a.pdf

[12] Gigaom: *10gen embraces what it created, becomes MongoDB Inc.* [Online; navštíveno 3.5.2017].
URL https://gigaom.com/2013/08/27/10gen-embraces-what-it-created-becomes-mongodb-inc

[13] Halvorsen, H.-P.: *Structured Query Language*. [Online; navštíveno 3.5.2017].
URL http://home.hit.no/~hansha/documents/database/documents/Structured%20Query%20Language.pdf

[14] Hart, P. E.; Nilsson, N. J.; Raphael, B.: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics, 1968, p.100–107.
URL http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf

[15] Jr, L. R. F.: *Network flow theory*. RAND Corporation, 1956, p-923.

[16] Khuller, S.; Raghavachari, B.: *Graph and Network Algorithms*. [Online; navštíveno 05.01.2017].
URL http://www.utdallas.edu/~rbk/papers/graphAlg.pdf

[17] Kovár, M.: *Studijní opora: Diskrétní matematika*. 2003.

[18] Lee, C. Y.: *An Algorithm for Path Connections and Its Applications*. IRE Transactions on Electronic Computers, 1961, p. 346-365, [Online; navštíveno 05.01.2017].
URL http://ieeexplore.ieee.org/document/5219222

[19] MongoDB: *MongoDB*. [Online; navštíveno 3.5.2017].
URL https://docs.mongodb.com/manual/introduction/

[20] Moore, E. F.: *The shortest path through a maze, in: Proceedings of an International Symposium on the Theory of Switching*. Harvard University Press, 1959, p. 285-292.

[21] Morrill, D.: *Announcing the Android 1.0 SDK, release 1*. 2008, [Online; navštíveno 15.5.2017].
URL https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html

[22] Odersky, M.; Spoon, L.; Venners, B.: *Combining Scala and Java*. [Online; navštíveno 3.5.2017].
URL http://www.artima.com/pins1ed/combining-scala-and-java.html

[23] Realm-Team: *Introducing Realm*. 2014, [Online; navštíveno 15.5.2017].
URL https://news.realm.io/news/introducing-realm/

[24] Reilly, S.: *NoSQL, MongoDB, and the importance of Data Modelling*. [Online; navštíveno 3.5.2017].
URL https://www.equalexperts.com/blog/tech-focus/nosql-mongodb-and-the-importance-of-data-modelling

[25] Strauch, C.: *NoSQL Databases*. [Online; navštíveno 3.5.2017].
URL http://www.christof-strauch.de/nosqldbs.pdf

[26] Wilson, R. J.: *Introduction to Graph Theory*. Longman Group Ltd, 1996, ISBN 0-582-24993-7, [Online; navštíveno 05.01.2017].
URL http://www.maths.ed.ac.uk/~aar/papers/wilsongraph.pdf

Prílohy

Príloha A

Obsah CD

Priložené CD obsahuje nasledujúce adresáre:

- `lyra` - zdrojové súbory serverovej časti
- `lyra-android` - zdrojové súbory klientskej časti (Android)
- `tex` - LaTeX zdrojové súbory tejto práce

Príloha B

Formát výstupu web API

platform

```
{
    "id": 1566,
    "name": "Semilasso",
    "zone": 101
}
```

vehicle

```
{
    "id": 12,
    "lineId": 225,
    "isBarrierLess": true,
    "lastPlatformId" : 1718,
    "latitude" : 49.224608,
    "longitude" : 16.583108,
    "delay" : 3

}
```

departure


```
{
    "platformId": 1104,
    "name": "Edisonova",
    "stopDepartures": [
        {
            "stopId" : 1,
            "description": ">Kolejni",
            "departures": [
                {
                    "lineName": "N99",
                    "isBarrierLess": false,
                    "finalPlatformName": "Technologicky park",
                    "timeMark": "01:12"
                },
                {
                    "lineName": "N99",
                    "isBarrierLess": false,
                    "finalPlatformName": "Technologicky park",
                    "timeMark": "02:12"
                },
                {
                    "lineName": "N99",
                    "isBarrierLess": false,
                    "finalPlatformName": "Technologicky park",
                    "timeMark": "03:12"
                }
            ]
        }
    ]
}
```

route


```
{
    "source": "Zahrebska",
    "destination": "Dobrovskeho",
    "sourceId": 1777,
    "destinationId": 1088,
    "totalTime": 300,
    "transports": [
        {
            "lineId": 53,
            "vehicleId": 1085,
            "vehicleType": "bus",
            "delay": 0,
            "entries": [
                {
                    "sequenceNumber": 1,
                    "arrival": -1,
                    "departure": 51720000,
                    "realArrival": -1,
                    "realDeparture": 1494332520000,
                    "name": "Zahrebska",
                    "platformId": 1777,
                    "lat": 49.219845,
                    "lon": 16.580415,
                    "description": ">Skacelova"
                },
                {
                    "sequenceNumber": 2,
                    "arrival": 51840000,
                    "departure": -1,
                    "realArrival": 1494332640000,
                    "realDeparture": -1,
                    "name": "Skacelova",
                    "platformId": 1567,
                    "lat": 49.22163,
                    "lon": 16.585935,
                    "description": "bus>Slovan.n."
                }
            ]
        },
        {
            "lineId": 12,
            "vehicleId": 1255,
            "vehicleType": "tram",
            "delay": 0,
            "entries": [
```

```
                {
                    "sequenceNumber": 3,
                    "arrival": -1,
                    "departure": 51960000,
                    "realArrival": -1,
                    "realDeparture": 1494332760000,
                    "name": "Skacelova",
                    "platformId": 1567,
                    "lat": 49.221651,
                    "lon": 16.585163,
                    "description": "Tram>mesto"
                },
                {
                    "sequenceNumber": 4,
                    "arrival": 52020000,
                    "departure": -1,
                    "realArrival": 1494332820000,
                    "realDeparture": -1,
                    "name": "Dobrovskeho",
                    "platformId": 1088,
                    "lat": 49.218351,
                    "lon": 16.587423,
                    "description": ">mesto"
                }
            ]
        }
    ]
}
```